University of Alberta

DESIGN AND IMPLEMENTATION OF AN OBJECT DATABASE FOR INJURY SURVEILLANCE

by

Adriana Mañas ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 1997

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-22632-8

Canada

University of Alberta

Library Release Form

Name of Author: Adriana Mañas

Title of Thesis: Design and Implementation of an Object Database for Injury Surveillance

Degree: Master of Science

Year this Degree Granted: 1997

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Adriana Mañas
11144 28 Avenue Apt.303
Edmonton, Alberta.
Canada T6J 4M2

Date: 10/June/1997

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Design and Implementation of an Object Database for Injury Surveillance** submitted by Adriana Mañas in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . .
Prof. M. Tamer Özsu (Supervisor)

. . . .
Prof. Louis Hugo Francescutti (External)

. . . .
Prof. Ling Liu (Examiner)

Date: 10 June 1997

To my husband, Alberto
who helps me achieve all my dreams.

Abstract

Injury is one of the most under-recognized public health problems. Reduction of injury is more likely to occur if data are available on causative factors, circumstances and populations at risk. This requires timely collection of data, their organization to enable cross-referencing and access, and the dissemination of these data in a way that is useful to health providers and researchers. The *Dynamic Injury Data Project (DIDP)* addresses these issues as a collaborative effort between the Department of Public Health (Faculty of Medicine and Oral Health), the Department of Computing Science (Faculty of Science) and the Faculty of Business. The objective of the project is to develop a system that will capture and link real-time data from emergency medical services, hospital, police, fire, utilities and administrative sources to facilitate studies in trauma outcomes research, medical quality improvement and injury prevention strategies.

There are essentially two components to the system being developed. The *Data Collection* component utilizes pen-based hand-held computers to be employed by the emergency medical services and hospital personnel to capture the most important patient-related information encompassing all pre-hospital, hospital and rehabilitative care. The *Database Server* component of the system stores the collected data and allows sophisticated analysis of the data. This thesis deals with the analysis, design and implementation of an object oriented database server for the DIDP. The server provides persistent storage of the data, ensure its integrity, and provide a mechanism for the applications to interact with the data.

Acknowledgements

First, I thank my husband, Alberto, for all his love, patience and understanding. I also thank my family in Argentina for all their support and encouragement. Specially I want to thank my mom, my dad, Maria Eugenia and Lito for the thousands of e-mails that made me feel so close to them. Thanks also to Cecilia, who is not only my sister but my best friend.

I also want to take this opportunity to thank my supervisor, Dr. Tamer Özsu, for his guidance and funding, and Dr. Louis Hugo Francescutti for his insight in injury surveillance and his continuous enthusiasm and support during the course of my thesis. Thanks also go to Dr. Duane Szafron for his advice in object oriented concepts. I am also grateful to Dr. Ling Liu for taking the time to read and comment on this thesis, and to Dr. Jeff Pelletier who chaired my thesis defense.

Help from my colleagues in the Dynamic Injury Data Project, Trevor Strome and Linda Nykolyn, and from the members of the Database Research Group, is appreciated. I also would like to thank Sundari Voruganti for all her support, company and friendship. I also acknowledge support from the Department of Computing Science in the form of a teaching assistantship and the M.Sc. Research Award.

Contents

1 Introduction **1**
  1.1 The Dynamic Injury Data Project 1
  1.2 Motivation 2
  1.3 Scope of the Thesis 3
  1.4 Thesis Organization 4

2 Background **6**
  2.1 Medical Informatics 6
  2.2 Public Health Surveillance Systems 6
    2.2.1 Utilization of Surveillance Data 7
    2.2.2 Planning a Surveillance System 8
    2.2.3 Sources of Surveillance Data 9
  2.3 Injury Surveillance Systems 11
    2.3.1 The Facts 12
    2.3.2 Related Work 12
    2.3.3 The Challenge 14
  2.4 Object-Oriented Database Systems 15
    2.4.1 Mandatory Features 16
    2.4.2 Optional Features 21
    2.4.3 Open Choices 22
  2.5 Summary 23

3 The Booch Methodology **24**
  3.1 The Notation 24
    3.1.1 Class Diagrams 25

3.1.2 Object Diagrams 29
3.1.3 Interaction Diagrams 30
3.1.4 State Transition Diagrams 31
3.1.5 Module Diagrams 31
3.1.6 Process Diagrams 31
3.2 The Methodology 31
3.2.1 Requirements Analysis 32
3.2.2 Domain Analysis 32
3.2.3 System Design 36
3.3 Summary 40

4 Requirements Analysis **41**
4.1 Patient's Flow 42
4.2 System Architecture 44
4.2.1 Data Processing Architecture 45
4.3 Database Requirements 47
4.3.1 The Database Server Charter 48
4.4 Summary 49

5 The Design **50**
5.1 Design Tool - Rational Rose 50
5.2 The Model 51
5.2.1 General Classes 51
5.2.2 Patient Identification and Health Information 55
5.2.3 Visits Information 57
5.2.4 Medications, Antibiotics and IVs 59
5.2.5 Diagnostic Images and Lab Exams 61
5.2.6 Invasive Therapy, Instrumentation and Fluids 61
5.2.7 Incidents and Personnel 64
5.2.8 Other Assessments 67
5.2.9 Gastrointestinal Assessment 69
5.2.10 Central Nervous System Assessment 69

5.2.11 Respiratory Assessment 71
5.2.12 Vital Signs Assessment 75
5.2.13 OR Anaesthesia and Procedures 77
5.2.14 EMS Specific Information 79
5.3 Summary 81

6 Implementation Issues **82**
6.1 Overview 82
6.2 Database Roots and Extents 84
6.3 Basic and Extended Types 85
6.4 The Class Interfaces 87
6.4.1 Object Creation and Validation 87
6.4.2 Object Deletion 90
6.4.3 Attribute Retrieval and Modification 90
6.4.4 Classes with Extent 91
6.4.5 Relationships 92
6.4.6 Printing Methods 93
6.5 Other Issues 93
6.5.1 The Visits Hierarchy Problem 93
6.5.2 Static Functions in C++ 95
6.5.3 Cascade Deletion 96
6.6 Summary 96

7 Conclusions and Future Work **97**

A Class Specifications **104**

List of Figures

3.1 Class Category 25
3.2 Class 26
3.3 Association Relationship 26
3.4 Uses Relationship 26
3.5 Has Relationship 27
3.6 Inherits Relationship 27
3.7 Abstract Adornment 28
3.8 Class Cardinality 28
3.9 Relationship Cardinality 29
3.10 Object 29
3.11 Link 30
3.12 Message 30

4.1 Patient's Flow 43
4.2 System Architecture 45
4.3 Data Processing Architecture 46

5.1 DIDP Main Diagram 52
5.2 General Classes 54
5.3 Patient Identification and Health Information 56
5.4 Visits Information 58
5.5 Medications, Antibiotics and IVs 60
5.6 Diagnostic Images and Lab Exams 62
5.7 Invasive Therapy, Instrumentation and Fluids 63
5.8 Incidents and Personnel 65
5.9 A possible solution for the critical incident problem 66

5.10 Other Assessments 68
5.11 Gastrointestinal Assessment 70
5.12 Central Nervous System Assessment 72
5.13 First approach in the design of spinal precautions 73
5.14 Respiratory Assessment 74
5.15 Vital Signs Assessment 76
5.16 OR Anaesthesia and Procedures 78
5.17 EMS Specific Information 80

6.1 Sample application program 83
6.2 didpInit(): how the database root is created 84
6.3 A method for retrieving the extent of a class 86
6.4 An insert() method for the class Color 88
6.5 A valins() method for the class Color 89
6.6 The visits hierarchy 94

Chapter 1

Introduction

Injury is one of the most under-recognized public health problems in the world today [Fra97]. Three and a half million people die, and seventy eight million are disabled every year as a result of injuries. Injury is the fifth leading cause of death for all age groups, and the leading cause for individuals under forty four years of age in Canada and the United States. More children over the age of one die from injuries than from cancer, heart disease, respiratory disorders, diabetes or Acquired Immune Deficiency Syndrome (AIDS).

Most injuries are not only predictable but preventable [FSH91]. Reduction of injury is more likely to occur if the causative factors, circumstances and populations at risk are known. These data can be used in planning, developing and implementing intervention strategies.

1.1 The Dynamic Injury Data Project

The *Dynamic Injury Data Project (DIDP)* is a collaborative effort between the Department of Public Health (Faculty of Medicine and Oral Health), the Department of Computing Science (Faculty of Science) and the Faculty of Business. The goal of the project is to develop a system that will capture and link real-time data from emergency medical services, hospital, police, fire, utilities and administrative sources to facilitate studies in trauma outcomes research, medical quality improvement and injury prevention strategies.

When fully implemented, the system will permit sophisticated analysis of injury trends focusing on the groups of people who are injured, the causes and circumstances, and the geographical locations of the incidents. The health community will have access to a multitude of data previously impossible or extremely difficult to integrate into analysis of injury occurrence and causes. The treatment data of the injuries will provide information regarding clinical practice guidelines by unveiling which treatments and procedures result in the best outcomes (e.g. shorter stay in hospital, less time off work). This information can then be used for medical quality improvement initiatives. Furthermore, diagnostic imaging and surgical interventions information, coupled with computer reconstruction and simulation techniques, can contribute to studies of the biomechanics of injuries. This potential for analysis will help researchers in synthesizing many more types and sources of data into a larger picture of the causes of injury and more insight into effective injury prevention interventions. Furthermore, electronic collection of data could allow sophisticated analysis by means of data mining techniques that would reveal unpredictable patterns and previously unrecorded associations.

There are essentially two components to the system being developed. The *data collection* component utilizes pen-based hand-held computers to be employed by the emergency medical services and the hospital personnel to capture the most important patient-related information encompassing all pre-hospital, hospital and rehabilitative care. The *database server* component of the system stores the collected data and allows sophisticated analysis of the data.

This thesis deals with the analysis of the system requirements and the design and implementation of the server component of the system.

1.2 Motivation

Most injury surveillance systems today rely on the analysis of routinely collected mortality and hospitalization data. Such systems have a number of deficiencies. First, these data are collected for administrative purposes only and do not contain detailed data needed for injury surveillance. For example, the circumstances in which certain injuries occur are very important in order to develop injury prevention strategies but are not relevant for administrative purposes and thus most of the time are not collected. Second, it is difficult for the research community to access such data.

Usually the data are only stored in paper form which makes it difficult and time consuming to analyze. In the few cases in which the data are stored electronically, those systems reside in a central office where the bureaucracy needed to access it is such that the data become outdated by the time the researchers get it, if they get it at all.

Despite the numerous recommendations by a variety of stake-holders, little progress has been made in the development of efficient injury surveillance systems [FSH91]. There is a demonstrated need in public health today for the development of such new, accurate, timely and accessible systems [GRT+94].

The DIDP system will meet the criteria for evaluation of surveillance systems proposed by the Center for Disease Control and Prevention in the United States [Eval88]:

- **Simplicity**. The design should be simple and the resulting system should be easy to use.
- **Flexibility**. The system should accommodate changes in information needs and operating conditions.
- **Acceptability**. The system should have the features that will encourage individuals and organizations to use the system on a regular basis.
- **Timeliness**. The system should provide up-to-date information at any time.

The DIDP system will collect data from the scene of the injury all the way through the time of the patient discharge and integration into the society. These data will not only consist of regular formatted data but also of *multimedia data* as well. Multimedia refers to the integration of structurally formatted data, textual descriptions, images, audio and video.

1.3 Scope of the Thesis

This thesis deals with the analysis, design and implementation of an object-oriented database server for the Dynamic Injury Data Project. The server will provide persistent storage of the data, ensure its integrity, and provide a mechanism for the applications to interact with the data. The major contributions of this thesis are:

- The analysis of the requirements necessary to develop the DIDP database server meeting the criteria mentioned earlier and the definition of the required data sets for injury surveillance.
- The definition of the system and data processing architectures for the DIDP system. These are initial architectures that could be easily adapted in the future if needed.
- The design of a detailed object-oriented model for the DIDP database. The model is general enough to be implemented in any object-oriented database system. The object-oriented approach was chosen because of its superior capability to represent multimedia data.
- The design of the database server mechanisms to ensure the consistency and encapsulation of the data.
- The implementation of the database server in the form of a class library to enable access of application programs and end users to the data.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 gives the background necessary to understand the rest of this thesis. Concepts in medical informatics, public health and injury surveillance systems, and object-oriented databases are explained. Chapter 3 gives an overview of the Booch Methodology, which is the software development methodology used by the DIDP project to develop the database server. Chapter 4 describes the requirements analysis. The flow of a patient through the real system, and the system and data processing architectures are described. The database server scope and responsibilities are also specified in this chapter. The design of a precise object-oriented model for the DIDP database is presented in Chapter 5. Implementation details of the server are addressed in Chapter 6. A detailed description of the class interfaces is provided. Some problems that arose during the project due to the implementation tools used, as well as their solutions are also discussed. Conclusions and future work are presented in Chapter 7.

Chapter 2

Background

2.1 Medical Informatics

Medical informatics refers to the application of information technology to enhance the quality of health care. Drs. Greenes and Shortliffe formally define *Medical Informatics* as "... the field that concerns itself with the cognitive, information processing, and communication tasks of medical practice, education and research, including the information science and the technology to support these tasks." [GS90]. They point out that although medical informatics is an intrinsically interdisciplinary field with a highly applied focus, it also addresses a number of fundamental research problems and planning and policy issues.

Medical informatics relies on computers to provide improved patient care. With new medical informatics applications and improved communication technology, health care providers will be able to access their hospital-based data and move across networks to other departments, institutions and data sources. They will be able to access information when, where and how they need it. As Ball and Douglas state "health care informatics is not a wildly futuristic vision. It is an evolving discipline now. In the 1990s, health care will realize the promise of [medical] informatics in education, research, administration and patient care." [BD90].

2.2 Public Health Surveillance Systems

Public health surveillance is "... the ongoing systematic collection, analysis and interpretation of outcome-specific data for use in the planning, implementation and evaluation of public health practice." [TC94]. A *surveillance system* has the capacity for data collection, analysis and timely dissemination of the analyzed information to persons responsible for the development of prevention and control programs.

2.2.1 Utilization of Surveillance Data

Public health surveillance data are used to assess the public health needs of the community, to evaluate existing programs, and to conduct research. The data show what the problems are, who is affected and where prevention activities should be directed. These data can also be used to evaluate the effectiveness of existing prevention programs, and to help researchers in identifying areas of interest for further investigation. The most important questions the data should be able to answer are *who*, *where* and *when*. According to Thacker [TC94] the uses of surveillance data include:

- **Quantitative estimates of the magnitude of a health problem.**
- **Portrayal of the natural history of a health problem.**

  Surveillance data can show how the health problem evolved indicating the different rates and populations affected.
- **Detection of epidemics.**

  Epidemics are not detected by analysis of routinely collected data but are identified through the alertness of the health providers. The existence of a surveillance system permits the conveyance of the information to give a prompt response to the problem.
- **Documentation of the distribution and spread of a health event.**

  The geographic patterns of a health event can help in trying to identify the causative factors.
- **Facilitating epidemiologic and laboratory research.**

  The identification of the populations at risk can lead to further epidemiologic and laboratory research, sometimes using the individuals identified as subjects in those studies.

- **Testing of hypothesis.**
  Surveillance data could be used to determine whether or not a particular action (e.g. a national vaccine program) yield the expected results.

- **Evaluation of control and prevention measures.**
  With routinely collected data, health officials can examine the effect of health policies.

- **Monitoring of changes in infectious agents.**
  The ability to monitor the changes of infectious agents permits health officials to facilitate prevention activities including notifying clinicians about proper treatment procedures.

- **Monitoring of isolation activities.**
  When suspicion arises of a possible spread of a serious disease, quarantines can be imposed. The people on quarantine would be monitored for a certain amount of time to ensure that the spread of the disease would not occur.

- **Detection of changes in health practice.**
  The detection of changes in health practice can lead to further investigation to learn the cause of the change and to study the impact of the change in the outcomes and costs associated with health care.

- **Planning.**
  With knowledge about the changes in the population structure and the conditions that affect them, health officials can plan to optimize the available resources for health care.

2.2.2 Planning a Surveillance System

The first step towards establishing a surveillance system is to have a clear understanding of what is expected from the system. A public health surveillance system may be established to meet a variety of objectives including assessment of the health status, establishment of public health priorities, evaluation of programs, and conduct of research.

Surveillance systems monitor the occurrence and outcomes of health events such as injury and disease. They monitor the frequency of the illness or injury, usually measured in terms of number of cases, incidence or prevalence; the severity of the condition, measured in case-fatality ratios, mortality, hospitalization and disability rates; and the impact of the condition, measured in terms of cost. Surveillance systems can also be utilized to monitor risk factors associated with certain illnesses or injuries and are used to monitor treatments that are the direct result of certain health events.

There are many health events that could potentially be tracked using a surveillance system. However, it is impossible to develop a surveillance system for every possible health problem. Ideally, resources will be allocated to the development of surveillance systems to monitor "high priority" health events. Although not an exact science, there are several indicators that are useful for identifying high priority health events such as frequency, severity, costs (direct and indirect), preventability, communicability, and public interest.

Once the purpose and need of a surveillance system has been identified, methods for obtaining, analyzing, disseminating and using the information should be determined and implemented.

2.2.3 Sources of Surveillance Data

This section describes the characteristics of five types of health information sources in which data is collected routinely and is generally available for analysis. As more information sources become available, effective surveillance for a specific health event will rely on the analysis and synthesis of information from a variety of sources, each of which have different strengths and limitations.

Notifiable Disease Reporting

Reporting on notifiable diseases at the national level started in many countries over one hundred years ago. The list of diseases for which notification is recommended has changed over time, and although there is overlap, the list varies from country to country and from region to region.

The results of the reports are collated and published nationally, but its primary purpose is to direct local prevention and control programs. The problem with this reporting mechanism is that, although many diseases or conditions are considered notifiable, compliance is poor in many countries and sanctions are rarely enforced.

In spite of its limitations, surveillance systems based on reporting of notifiable conditions are a mainstay of public health surveillance. Unlike other sources of routinely collected data, information from notifiable conditions is available quickly and from all regions. In the future, reporting of notifiable conditions will be based on computerized databases developed for billing or other purposes. However, the utility of these systems is limited at present as they do not usually used the standard International Classification of Disease (ICD) codes.

Vital Statistics

Data collected at the time of birth and at the time of death is one of the cornerstones of surveillance. Vital statistics are an important source of information as it is the only health-related data available in many countries in a standard format. More than 80 countries report vital events to the World Health Organization coded and tabulated according to the International Classification of Diseases (ICD).

The usefulness of vital statistics for surveillance of a particular health event depends on the characteristics of the event and the procedures used to analyze the data. Although birth and death certificates are issued shortly after the event, results of processing the data and producing a final report at a national level can take several years [TC94].

In spite of the limitations, vital statistics are an important source of information for surveillance at the local, national and international level. Although differences in rates do not always reflect differences in disease and injury, routine analysis of the birth/death information can highlight areas where further investigation is necessary.

Registries

Registries differ from other data sources for surveillance in that information from multiple sources are linked together for each individual over time. These sources include hospital discharge reports, treatment records, pathology reports and death certificates. Information from registries have been widely used for research purposes, but in many cases they have also been used for surveillance and related activities.

The most successful registries are those that have realistic purposes and where the collected data are accurate and limited to essential information. Even when data collection appears to be straight-forward, the time and resources required to develop a registry are often underestimated.

Surveys

Surveys can provide useful information in assessing prevalence of health conditions and potential risk factors, and for monitoring the changes in prevalence over time. They are also used to assess knowledge, attitudes and health practices in relation to certain conditions. The people surveyed are usually queried once and are not monitored individually after that. Surveys can be conducted through questionnaires and personal or telephone interviews. The survey sample has to be representative of the source population to provide representative results.

Administrative Data Collection Systems

Administrative information that is routinely collected about episodes of care (e.g. hospitalizations, visits to emergency rooms and health care providers, etc.) can also be used for surveillance purposes. In most cases these data are computerized for billing purposes only, but since they include diagnosis information, they can also be used for surveillance. Data that include personal identifiers is important so that statistics can be calculated on the basis of persons rather than on episodes of care. Special precautions are needed to ensure the confidentiality of the individuals whose identifiers are stored in the computerized data. Although most administrative data are available only for certain types of health care (e.g. hospitalizations), analysis of administrative data is useful for public health surveillance and program planning.

2.3 Injury Surveillance Systems

Surveillance systems for infectious diseases have existed for decades. Although injuries have long been identified as a major public health problem, surveillance systems that monitor and control injuries are only in their infancy [TC94].

An injury is "any specific and identifiable bodily impairment or damage resulting from acute exposure to thermal, mechanical, electrical, or chemical energy, or from the absence of essentials such as heat or oxygen." [Fra97]. Many people and many physicians regard injuries as accidents [FSH91]. Injuries are nor accidents; accidents are random events while injuries are predictable and preventable [FSH91].

2.3.1 The Facts

Accurate and timely information is the cornerstone of effective injury prevention and control. Yet, up to 97 percent of all injuries that require medical attention are never recorded in any comprehensive data set for use in injury surveillance [WFP96]. Garrison et al [GRT+94] claims that the failure to record these potential surveillance data impedes assessment of a community's health care needs.

Mortality data has been extensively used to design injury prevention programs, but deaths are just a small part of the problem. It has been estimated that for every injury death, there are as many as 330 visits to the hospital emergency departments [RBB92]. Other studies show that injury deaths account for less that 0.2 percent of all injured patients [WFPP95] and sometimes even less than 0.1 percent [RRTB92].

Hospital discharge data has also been used for injury surveillance, but the lack of information about external cause of injury limits its usefulness for prevention planning [RBB92]. Furthermore, the information provided by these discharge reports only account for less that 2.5 percent of the injured population each year [WFPP95].

Data for emergency departments have been used in several major studies [RBB92]. But, unlike hospital discharges, there is no standardized reporting system and the type of data obtainable from emergency departments is not well documented. Furthermore, this information is not usually stored at the emergency department and is only retrievable searching through the hospital paper records, which can be a cumbersome procedure.

2.3.2 Related Work

To solve the lack of information on causal factors of injuries, Ribbeck et al [RRTB92] proposed a method to assign external cause of injury codes (E-codes, a subset of the International Classification of Disease, 9th revision codes) to all injury patients seen in a large volume emergency department. An E-code assignment sheet was designed for use by the triage nurse of the emergency department. This sheet contained a checklist with the frequently occurring codes of injury. The registered nurse at the triage desk recorded the cause of injury as the patient first encounter as well as the chief complaint, vital signs and treatment priority. This study demonstrated the feasibility of collecting data on causal factors of injuries in large emergency departments without much difficulty.

Because 80 to 90 percent of all injured persons that seek medical attention are cared for at emergency departments [WFPP95], most efforts have focused on developing injury surveillance systems based on data generated by these departments. Garrison et al [GRT+94] identify emergency department surveillance as a way of documenting illness and injury patterns and for responding to health care challenges in the community. In their work they examine the overall concept of emergency department surveillance and related issues, and propose a national strategy for implementing this type of surveillance.

Runyan et at [RBB92] have conducted a study to determine the routine record-keeping practices in hospital emergency departments to assess the adequacy of these information for injury surveillance and prevention planning. The study demonstrates that the type of data collected in different emergency departments vary considerably. It also shows the absence of information about the external causes of injury. The authors conclude that efforts to standardized the record-keeping process would enhance any use of emergency data. They also point out that the development of methodologies to secure the necessary information has potential not only for surveillance and research but to ease the burden of record-keeping among busy clinicians and to help hospitals with concerns about quality of care and cost reimbursement.

Williams et al [WFPP95] develop an injury surveillance system based on the emergency department log. The emergency log was modified for the collection of injury related data such as the external cause of injury. A list of injury causes was developed on the basis of review of existing literature and pilot tests. The log data were entered into a computerized database, and descriptive analysis was performed. The list of injury causes was successful in 93 percent of the emergency department cases during the pilots. The authors conclude that the expansion of emergency department logs for collection of injury data require minimal training and costs and provide an excellent source for injury surveillance.

Williams et al [WFP96] extend their work by proposing a more complete emergency based injury surveillance system. The idea was to link the emergency department logs with the existing computerized hospital records. They created a system to merge the files by the hospital identification number and the date of service as their key merge variables. Although some problems were encountered, more than 97 percent of the patients seen in the emergency department had additional data after the merge. With this system, significantly more data can be examined to help in injury prevention and planning.

Many efforts have also been focused on the development of trauma registries [TC94]. Trauma is defined as "blunt or penetrating injuries or burns" [PM89]; this definition excludes other types of injuries such as poisonings, asphyxiations, immersion or exposures to extreme temperatures. Some professionals claim that data related to all injuries can not be captured as most hospitals do not have sufficient resources for this task [PM89]. They also claim that trauma registries can serve as a principal tool for the systematic audit of patient care provided by hospitals and trauma centers and as a potential source of part of the data needed by injury surveillance [PM89].

2.3.3 The Challenge

Development of an injury surveillance system is not an easy task. Some of the important issues that need resolution include defining the data set to be captured and defining the injured population. Currently, there is no standard and easy accessible mechanism to obtain information on incidence, demographics, type, and cause of injury. To be effective, a surveillance system must achieve a balance between completeness and practicality.

Although some efforts have been made to implement better injury surveillance systems, they have two major drawbacks. First, the data collection is only focused in emergency departments, not considering other data generated in emergency medical services or the rest of the hospital. Second, the data is collected in paper form and subsequently entered in some type of database. This methodology duplicates the effort (and costs) of collecting data, delays the availability of the data to be used by the health care providers during the patient visits to the hospital, and introduces the possibility of errors and/or loss of data.

There is a demonstrated need in public health for the development of new, accurate, timely and accessible injury surveillance systems. The DIDP challenge is to create a system that:

- collects all the data needed for injury surveillance;
- provides the latest computer technology;
- facilitates the health providers' job in the data collection;
- makes data available at the same time in which is being collected; and
- helps in administrative tasks, eliminating the need to enter the same data more than once, and thus, preventing errors.

2.4 Object-Oriented Database Systems

Database management systems (DBMS) provide many advantages over the traditional file processing approach [EN89]. They have been used for many years in many areas including business, engineering, medicine, law, and education, to name a few. Not only do they provide standard services such as data abstraction, support for multiple users, redundancy control, access control, backup and recovery, but they also provide the ability to go beyond the simple retrieval of information to high-level access through a powerful query mechanism.

The DIDP system, when fully implemented, will not only consist of regular formatted data, but of multimedia data such as image, audio and video as well. Since the relational model, which is the currently the most popular model for a variety of applications, has difficulties in representing the complex data present in multimedia applications [Sch96], an *object-oriented DBMS* was chosen for the DIDP project.

In the following sections we present the main features and characteristics of object-oriented DBMSs. These features are separated into three groups following the organization of Atkinson's et al [ABD+89] "Object-Oriented Database System Manifesto". The groups are:

- **Mandatory features** that a system must comply with in order to be termed object-oriented DBMS;

- **Optional features** that can be added to improve the system, but which are not mandatory; and,

- **Open choices** that the designer selects from a number of options.

2.4.1 Mandatory Features

An object-oriented DBMS must have the characteristics of a DBMS and of an object-oriented system [ABD+89]. To be a DBMS it must support persistence, secondary storage management, concurrency, recovery and an *ad hoc* query facility. In order to be an object-oriented system it must support complex objects, object identity, encapsulation, types or classes, inheritance, overriding and late binding, extensibility and computational completeness.

Persistence

Persistence is the ability to make the data survive past the execution of a process in order to be reused by another process. The persistence should be *orthogonal* to the type of data, that is, each object must be able to become persistent independently of its type and without any explicit translation. Also, the user should not have to move or copy the data to make it persistent.

Secondary Storage Management

This feature prevents the user from having to write code to manage certain aspects of the physical level of the system. Secondary storage management is a classical feature of DBMSs that include index management, data clustering, data buffering, access path selection and query optimization. None of these are visible to the user; they are simply performance features. They provide a clear independence between the logical and physical level of the system.

Concurrency

The system must accommodate multiple users accessing the system at the same time. It should, therefore, provide a mechanism to ensure the atomicity of a sequence of operations (i.e. transactions) and controlled sharing of data.

Recovery

The system should provide a mechanism to recover from software or hardware failures; that is, the system should bring itself back to some coherent state upon recovery.

Ad Hoc Query Facility

The objective of the query facility is to allow the user to ask queries to the database declaratively. Atkinson et al [ABD+89] define three characteristics that a query facility should have:

- It should be able to express in a few words or mouse clicks non-trivial queries concisely. It has to emphasize the *what* and not the *how*.
- It should be efficient. It should have some form of query optimization.
- It should be application independent. It should work on any possible database, eliminating the need to write additional operations on user-defined types.

Complex Objects

Complex objects are built by applying constructors to simpler objects [BM93]. Simple objects are integers, real numbers, characters, variable length strings, and booleans. The minimal set of constructors that a system must provide include sets, lists and tuples. *Sets* are very important as they are a natural way of representing real world collections. *Tuple* constructors provide a natural means of representing attributes in an entity. *Lists* or *Arrays* are sets with ordering on the elements and are necessary in some applications where matrices or time series data are needed.

Object constructors must be orthogonal, that is, they should be applicable to any object. In the relational model constructors are not orthogonal as set constructors can only applied to tuples and tuples can only be applied to atomic values.

Object Identity

Every object must be identified with a single object identifier (OID), which must be independent of the values of the object attributes. By using OIDs, objects can share other objects, and a general object network can be built [BM93].

Because of this concept, two notions of equivalence exist: identity equality and value equality. Two objects are *identical* if they are the same object (i.e. the object identifiers are the same). Two objects are *equal* if the values of the attributes of both objects are recursively equal. This means that two identical objects are equal but the inverse is not necessary true.

Encapsulation

The idea of encapsulation comes from the necessity of clearly distinguish between specification and implementation, and the need for modularity. Modularity is an essential principle for designing and implementing complex software where a team of programmers is involved [BM93]. It is also important for supporting object authorization and protection mechanisms.

Encapsulation in programming languages derive from the concept of abstract data type where an object consists of an interface and an implementation. The *interface* is the specification of the operations that can be invoked on the object and it is the only visible part of the object. The implementation has a data and a procedural part. The data represent the state of the object, and the procedural part describes the implementation of each operation in some programming language.

Whether the structural part is part of the interface is a matter of debate in object-oriented DBMSs, while in the programming language approach the structure is clearly part of the implementation. Although proper encapsulation is achieved when data are part of the implementation, there are cases where encapsulation is not needed and the use of the system could be significantly simplified if encapsulation is violated in special circumstances [ABD+89]. For example, with *ad-hoc* queries, encapsulation can be eliminated as issues on maintainability are not important. Thus, an encapsulation mechanism must be provided by any object-oriented DBMS, but there are some cases where its enforcement is not appropriate.

Types or Classes

Object-oriented systems can be divided in two categories: those that support the concept of *type* and those that support the concept of *class*.

A *type* summarizes the common features of a set of objects with similar characteristics. This concept corresponds with the concept of abstract data type. A type has two parts: an interface and an implementation. The interface consist in a list of operations and their signatures, and it is the only visible part for the users of the type. The implementation consists of the data part, which describes the internal structure of the object's data, and the procedures that implement the operations defined in the interface. In programming languages, types are used to increase the programmer's productivity, ensuring the correctness of the programs. If the type system is carefully designed, the type checking is done at compilation time; otherwise it could be deferred until run-time. In type-based systems, types can not be modified in run-time.

A *class* specification is the same as that of a type, but it is more of a run-time notion [ABD+89]. It contains a *new* operator that allows the creation of new objects. Also, a class has its extension (i.e. the set of objects that are instances of the class) attached. The user can manipulate this extension by applying certain operations. Classes are not created to check the correctness of a program but to create and manipulate objects.

An object-oriented DBMS has to provide one of these two forms of data structuring. However, it is not necessary for the system to automatically maintain the extent of a type. Consider, for example, the **Date** type that can be used by many users in many databases. It does not make sense to automatically maintain the set of all the dates used in the system. On the other hand, in the case of a type such as **Patient**, it might be nice for the system to maintain the patient extent.

Inheritance

The concept of inheritance is the most important concept of object oriented programming [BM93]. With this mechanism a type called a *subtype* can be defined on the basis of the definition of another type called a *supertype*. The subtype inherits the attributes and behavior of its supertype. In addition, a subtype can have its own attributes and behavior which are not inherited. Inheritance provides code reuse and maintainability.

Overriding, Overloading and Late Binding

There are cases where one wants to use the same name for different operations. One common example is the *draw* operation. This operation takes an object as input and draws it on the screen. Different types of objects are displayed differently (e.g. a line, a circle, a rectangle). This forces the programmer to be aware of the type of the object in order to invoke the correct *draw* operation. For example, if a programmer wants to draw all the objects in a set whose type is unknown until run-time, in a conventional system, he/she would have to write:

```
for x in X do
   begin
      case of type (x)
         line: draw-line(x);
         circle: draw-circle(x);
         rectangle: draw-rectangle(x);
   end
end
```

In an object-oriented system, the *draw* operation is defined at the most general drawable type in the system, and then the implementation of the operation is *redefined* for each subtype according to the type necessity (this redefinition is called *overriding*). This results on a single name (draw) denoting more than one program (that is called *overloading*). Therefore, to draw the set of elements in the set, the programmer applies the draw operation to each of them, and the system will pick the appropriate implementation at run-time.

```
for x in X do draw(x);
```

Although the implementors would have to write the same amount of code, the application programmer does not have to worry about the different programs for drawing. Also, the code is simpler, as it does not have any *case* or *if* statements, and if another type is later added, the existing application programs do not need to be modified.

In order to support this feature the system can not bind operation names to programs at compile-time. They must be resolved at run-time. This delayed translation is called ***late binding***.

Extensibility

Every DBMS come with a set of predefined types. These types can be used by the programmer to write their applications. An object-oriented DBMS must be *extensible*, that is, it should provide a mechanism to define new types and there should be no distinction in usage between the system defined and the user defined types. However, it is not required that the collection of type constructors (tuples, sets, lists, etc.) be extensible.

Computational Completeness

An object-oriented DBMS is computational complete if one can express any computable function using the data manipulation language of the system. The most common way to introduce computational completeness is to provide a reasonable connection to existing programming languages [ABD+89]. Being computational complete, object-oriented DBMSs are more powerful than traditional systems which only store and retrieve data and perform simple computations on atomic values.

2.4.2 Optional Features

This group includes those features that improve the system, but which do not have to be included in an object-oriented DBMS. These features include multiple inheritance, type checking, distribution, design transactions and versions.

Multiple Inheritance

With single inheritance, each subtype has exactly one supertype. With multiple inheritance a type can have more than one supertype. Because there is no consensus among the programming languages regarding this issue, this feature is considered optional for object-oriented DBMSs.

Type Checking

The amount of type checking at compile time is left open. The more type checking that can be performed, the better it is since this will prevent run-time errors.

Distribution

The distribution of the database should be orthogonal to the object-oriented nature of the system. An object-oriented DBMS may or may not be distributed.

Design Transactions

In many new applications, the classical transaction model is not satisfactory: transactions tend to be long and the usual serializability criterion is not adequate. Many object-oriented DBMSs provide design transactions (i.e. long and nested transactions).

Versions

Versioning is a characteristic that many new applications such as CAD/CAM or CASE need. Thus, object-oriented DBMSs could provide a versioning mechanism to satisfy this need.

2.4.3 Open Choices

In this group we include those features where no consensus have been made by the scientific community and where the different approaches do not make a system more or less object-oriented.

Programming Paradigm

There is no reason to impose one programming paradigm over another. Any programming paradigm could be chosen for the system. One solution might be to make the system independent of the programming style and provide multiple programming paradigms. The choice of syntax is also free and people could argue forever which one to choose.

Representation System

The representation system is the set of atomic types and constructors provided by the system. Although there is a minimal set of atomic types and constructors that has to be provided, this representation system could be extended in many different ways.

Uniformity

The degree of uniformity of these systems is another open issue. At the implementation level one should decide whether type information should be stored as objects or not. This decision should be made based on the performance and ease of implementation. At the programming language level, one should decide if types are first class entities in the semantic of the language or not.

2.5 Summary

In this chapter, many concepts that are needed to understand the rest of this thesis are introduced. First, an overview on medical informatics and general concepts on public health surveillance systems are explained. Then, specific issues on injury surveillance systems and related work in the area are discussed. Object-oriented DBMSs are also presented at the end of the chapter.

Chapter 3

The Booch Methodology

Development of a complex system such as DIDP involves accommodation of many and diversified requirements. Using a software development methodology provides a standardized means of presenting and communicating the system requirements and design decisions. The Booch Methodology [Booch93] was chosen as the analysis and design methodology for the DIDP system.

The Booch Methodology is one of the most popular methodologies for object-oriented analysis and design. It provides an expressive notation and a set of heuristics needed by most businesses to produce working systems efficiently. It provides a model that allows developers to enhance, correct and maintain the same consistent model from the beginning of analysis through implementation. The advantage of having this unique model is that there is no throwaway work and the analysts and implementors can use the same specifications thus preventing misunderstandings. Changes to the system only have to be done in this unique model instead of changing multiple models and documents which greatly simplifies the maintenance of the system. The model and the system evolve together providing updated documentation at any time.

This chapter describes the Booch Methodology based on some parts of the structure of White and Goldberg's "Using the Booch Method: A Rational Approach" [WG94].

3.1 The Notation

As Booch mentions [Booch93]: "The fact that this notation is detailed does not mean that every aspect of it must be used at all times". He also emphasizes that the notation is not an end in itself and that one should apply only those elements of the notation that are needed and nothing more. In this section the focus is on a subset of the notation used in this project. For a complete description of the notation the reader can refer to Booch's book [Booch93].

The notation consists of six different diagrams that will be introduced in the following sections.

3.1.1 Class Diagrams

The class diagram is the core diagram in the Booch notation. It shows the existence of classes and their relationships. Class diagrams can contain class categories, classes or a combination of the two.

Class Category

Class categories (Figure 3.1) serve to divide the system into logical units. They are formed by a group of logically related classes that have low interaction with the classes of other categories. Class categories can also contain other class categories. For certain diagrams it is useful to show some of the classes contained by the class category. If we do not want to show any we can simply drop the separating line and only show the category name. Classes in a category might need classes that belong to another category. The *using* relationship icon is used to indicate such relationship (See "Uses Relationship" later in this section). If the category is used by all of the other categories we indicate it with the key word *global* in the category icon.

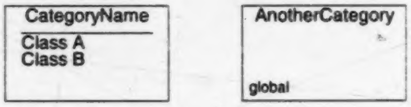

Figure 3.1: Class Category

Class

A class captures the common structure and behavior of a set of objects. It is an abstraction of a real-world entity. When one of these entities exists in the real world, it is an instance of the class and is called object. The major attributes, operations and constraints of the class can be specified inside the class icon (Figure 3.2). If we do not want to show any of these properties we can simply drop the separating line and only show the class name.

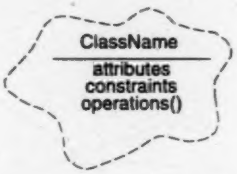

Figure 3.2: Class

Association Relationship

An *association* is a connection between two classes (Figure 3.3). It is the most generic type of relationship. Associations are always bidirectional.

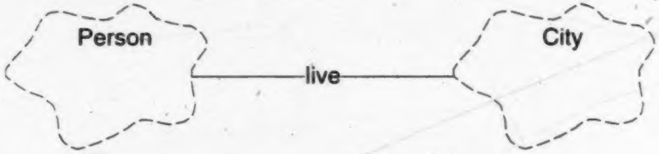

Figure 3.3: Association Relationship

Uses Relationship

A *use* relationship between two classes denotes that a service from the target class is being used by the source class, or that operations of the source class have signatures whose return class or arguments are objects of the target class (Figure 3.4). The class with the circle end of the relationship is the source and the other one is the target.

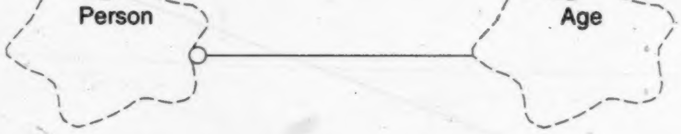

Figure 3.4: Uses Relationship

Has Relationship

A *has* relationship is used to show the whole-part relationship between two classes (Figure 3.5). It is also known as ***aggregation*** relationship. The class with the circle end of the has relationship is called the aggregate class. The class at the target end of the ***has*** relationship is the part whose instances are owned or contained by the objects of the aggregate class. The main difference with the *uses* relationship is that in the *has* relationship the aggregate objects own their parts. This means that when an object is deleted in the aggregate class all the objects that are owned by that object must be deleted as well, as they are just part of that object.

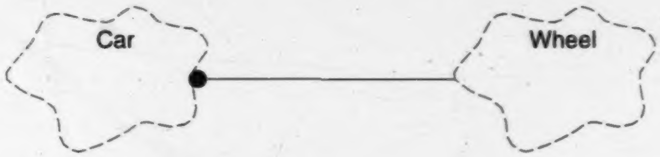

Figure 3.5: Has Relationship

Inheritance Relationship

An *inheritance* relationship is used between two classes to show a *is-a* relationship between them (Figure 3.6). When a class inherits from another class it means that it shares its structure and behavior. The arrowhead points toward the base class (or superclass). The other class is called the subclass.

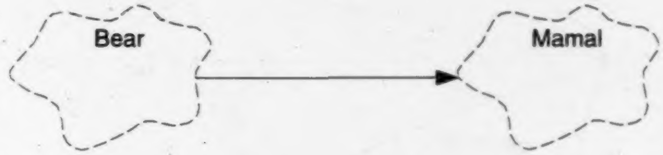

Figure 3.6: Inherits Relationship

Abstract Adornment

Abstract classes do not have instances and are used to define commonalities among a group of classes. In order to identify an abstract class a letter "A" inside a triangle is used (Figure 3.7).

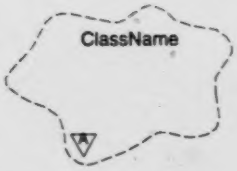

Figure 3.7: Abstract Adornment

Cardinality Adornment

Cardinality can be specified for classes and for relationships. The possible cardinality values are:

Value	Description
1	One instance
n	Unlimited number
0..n	Zero or more
1..n	One or more
0..1	Zero or one
<literal>	Exact number
<literal>..n	Exact number or more
<literal>..<literal>	Specified range

If cardinality is applied to a class it means that the class is only allowed to have that number of instances (Figure 3.8). If no cardinality is specified for a class the default value is n.

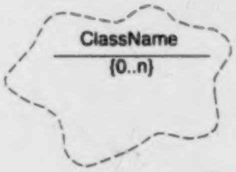

Figure 3.8: Class Cardinality

When cardinality is applied to a relationship it indicates the number of links between the instances of the source and the target class (Figure 3.9). There is no default value for unspecified relationship cardinalities.

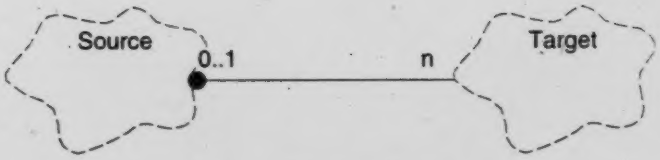

Figure 3.9: Relationship Cardinality

3.1.2 Object Diagrams

Object diagrams are used to show a snapshot in time of a transitory group of events over a certain configuration of objects. Each object diagram represents interactions or structural relationships that may occur among a certain number of objects.

Object diagrams are used to show the different *use cases* or *scenarios* and to understand the behavior of the system. Jacobson [Jac92] defines a *use case* as "a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction or sequence of interrelated events." In the DIDP system the action of dispatching an ambulance is an example of a use case.

The essential elements of an object diagram are *objects*, *links* and *messages*.

Object

Each object represents an instance of its class. The object icon is similar to the class icon except that it has a solid line as a boundary (Figure 3.10). If multiple instances of the same class are used the multiple objects icon can be used.

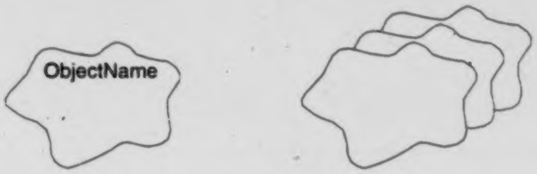

Figure 3.10: Object

Link

Objects interact with other objects through their links. A link is an instance of any relationship that exists between two classes (e.g. *has* or *uses* relationship). It is analogous to an object being an instance of a class. A link may exist between two objects only if the classes of those objects have a relationship between them. The existence of a link means that the objects can communicate: one can send messages to the other. An object can also send messages to itself; that means that an object may be linked to itself. The link is represented by a line between the objects (Figure 3.11).

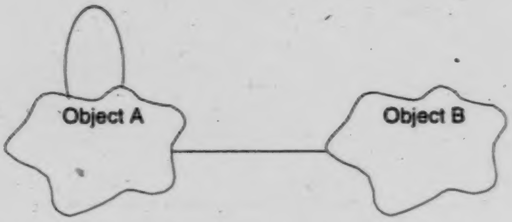

Figure 3.11: Link

Message

The message icon shows the direction in which a message is sent (Figure 3.12). Messages go generally in one direction but can also be bi-directional. To show the order of the events the messages can have a sequence number (starting at one). If no sequence number is specified the message can be transmitted at any time relative to all other messages.

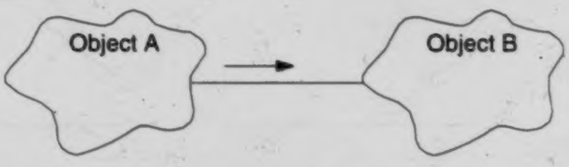

Figure 3.12: Message

3.1.3 Interaction Diagrams

Interaction diagrams are an alternative to object diagrams. They are also used to represent the scenarios or use cases. The relative order of the messages is easier to follow but they do not show links or attribute values as in object diagrams.

3.1.4 State Transition Diagrams

Each class can have associated a state transition diagram. These diagrams show the event-ordered behavior of the instances of that class. A state transition diagram should only be supplied for classes that have significant event-ordered behavior.

3.1.5 Module Diagrams

Module diagrams show how classes and objects are allocated to modules in the physical model of the system (i.e. the collection of directories and files that composed the system). The use of these types of diagrams depends on the implementation language since not all languages support this concept. For example Ada supports all the module types in the Booch Methodology, C++ only supports the concept of simple files and Smalltalk does not support the concept of module at all.

3.1.6 Process Diagrams

Process diagrams show how processes are allocated to the processors in the physical model. These diagrams are only needed if the process structure of the system needs to be represented.

3.2 The Methodology

In the past, many methodologies have used a rigid series of steps. These methodologies require the completion of one step before continuing with the next one. The problem with this approach is that in reality developers use an iterative process rather than a linear one. The Booch Methodology provides an iterative process based on three mini-steps. This means that the developers will do a little bit of analysis, a little bit of design, a little bit of coding, cycle back and do it again. All the steps of the methodology are completed but in a series of cycles instead of in large leaps. The Booch Methodology consists of three steps:

- Requirements analysis
- Domain analysis
- System design

3.2.1 Requirements Analysis

The objective of the requirements analysis is to determine what the customer wants the system to do. In this step the key functionality and the scope of the system domain must be defined. requirements analysis is a contract between the customer and the developers on what the system will do. It is not, however, a fixed contract. As the development goes on there might be changes, but the requirements analysis will serve as a starting point and a reference on what the system is supposed to do.

Use case analysis is a method to describe system functions. The collection of all the use cases of a system describes the complete functionality of such system. During this step the developers and the domain experts work together defining all the key use cases of the system. These use cases will be used later to define the classes and operations.

There are no formal steps for requirements analysis because this process can be very different from one situation to another. The only key for a good requirements definition is a good understanding of the problem domain and the customer needs.

Deliverables of Requirement Analysis

The methodology requires two formal products from the requirements analysis:

- **System charter**, which is a description of the responsibilities and the scope of the system.
- **System function statement,** which is the collection of the key use cases of the system.

3.2.2 Domain Analysis

The objective of the domain analysis is to define a precise object oriented model of the part of real world that is relevant to the system. During this step all data and major operations of the system are identified and added into the model. This process also solves all the problems of vocabulary that might arise in the requirements analysis and any contradicting requirements that might exist. Good communication skills play a key role for this step. The following steps are performed during domain analysis:

- Defining classes
- Defining relationships
- Defining operations
- Finding attributes
- Defining inheritance
- Validation and iteration

Defining Classes

The goal of this step is to identify the major classes of the system. The focus must be on identifying those classes that reveal more about the problem domain. The idea is not to obtain all the classes of the system but to get a starting point for the analysis. The Booch Methodology is a highly iterative process and as we get deeper into the analysis more classes will appear.

One way of discovering key classes is to find the nouns in the problem statement, as they often correspond to classes. Although this is a very useful trick it is important to maintain the focus at the logical level without paying attention to implementation characteristics. We have to concentrate in the *whats* and not in the *hows*. It is also important to avoid contextual information present in the problem statement that is irrelevant to the system responsibilities. We have to be careful with the ambiguity of the natural language as well.

Another important thing to keep in mind is the choice of meaningful names for the classes. A good name should bring to mind the abstraction that is being represented without any further knowledge. Names must be singular nouns (unless the class represents a collection) or an adjective and a noun. The inability to name a class means that something is not clear about that class or the abstraction we are trying to represent is not a class but something else (e.g. relationship, attribute).

During this step we also have to start building the project's *data dictionary*. A data dictionary is a central repository for the domain entities, including classes, attributes, relationships and operations, that are found during the analysis and design. As the key classes are discovered we add them into the data dictionary.

Once all the initial classes are defined, the semantics of these classes must be identified. This means that we have to provide a short definition for each class and also have to note any rules or constraints that we know about the class.

Defining Relationships

Classes are not isolated entities. They relate to each other to form the structure of the system. The goal of this step is to identify those relationships. There are two types of relationships we have to identify: ***association*** and ***aggregation*** (also known as *has* relationship). For an explanation of each of these relationships see Section 3.1.1.

It is important to choose meaningful names for the relationships. The name must provide significant semantic information. Names of relationships are usually noun phrases that denote the nature of the relationship. They do not need to be unique in the system, but only within their own context. The inability to name a relationship could be a sign of incompleteness or that more than one relationship was mistakenly gathered together. When it is difficult to define a relationship, further analysis might show that new classes and relationships need to be added. This refining of relationships is a very important step in the process.

Similarly, we also have to provide a short definition for each relationship capturing any rules or constraints that we know about. We also have to define the cardinality of the relationships. The cardinality shows whether it is mandatory that an instance of the source contains an instance of the target, and the maximum number of target objects that the source can contain at any one time. For more information on cardinalities see Section 3.1.1.

Defining Operations

The goal of this step is to identify the operations needed to support the system functions. In order to determine which operations are needed, each class should be examined. It is also during this step that the use cases defined in the requirement analysis are expanded into detailed scenarios using object diagrams. Modeling scenarios show how objects collaborate in the use case and identify the operations needed in each object.

Each operation should perform one simple function. Too many inputs and outputs should be avoided. This may indicate that the operation is doing too many things and it should be split into two or more simpler ones. Input switches also have to be avoided as they are often a sign of non-primitive functions. Choosing a representative name for the operation is also important. The name should reflect the outcome of the function. The operations must be added to the class specification including any information about the arguments and return value.

Finding Attributes

During this step the properties of the classes will be defined. The properties that describe a class are known as *attributes.* An attribute is like an aggregation relationship, where the label is the attribute name and the cardinality is exactly one.

In order to find these attributes each class must be examined. Unlike classes and relationships, attributes are often not mentioned in the problem statement. The knowledge about the problem domain has to be used to define them. Adjectives that describe the object are usually good candidates for attributes. We have to keep in mind the problem domain and do not add unnecessary attributes to the classes. The color of the eyes of a person can be an attribute of the class **Person**, but it might not be pertinent to the actual problem domain.

The attributes must be added to the class specification including their types. Any attribute that is derivable from other attributes, if specified, must be noted as derivable.

Defining Inheritance

The goal of this step is to generalize and specialize the classes with similar domain types. There are two ways of doing this:

- **Identifying superclasses.** When common data or behavior is found among a group of classes we can gather that common information in one class and make the other classes subclasses of this new superclass.
- **Identifying subclasses.** If a subset of attributes or operations only apply to a subset of the instances of the class it is useful to define a subclass.

Validation and Iteration

To validate the model we have to check that all the classes, relationships, operations and attributes that we have defined are sufficient to implement the system defined in the system charter specified during the requirement analysis.

There are two ways of validating the model: by using object diagrams and by checking key outputs. In both cases we have to pick one or more scenarios/key outputs respectively and check that we have enough information in the model to implement them.

Once the validation has been done we have to consider if we still have to iterate all the steps in order to refine the model or if we have finished with the domain analysis and can move to design. We can stop analysis and go to design when, after iterating and walked through all the scenarios, we have not found any more classes, relationships, operations or attributes. It should be remembered that this is a highly iterative process and we can come back to the domain analysis at any time.

Deliverables of Domain Analysis

The deliverables of domain analysis include:

- **Class diagrams** identifying the different classes of the system with their relationships.
- **Class specifications** for all the classes. They must include all the information regarding each class including its relationships, superclasses, attributes and operations.
- **Object diagrams** showing the use cases defined in the requirement analysis. Object diagrams or interactions diagrams can be used.
- **Data dictionary** listing all the entities in the domain including classes, relationships and attributes.

3.2.3 System Design

Analysis focuses on understanding the domain; design focuses on how to implement those requirements.

An iterative approach during design is even more important than during analysis. Trying to make a complete design in one step is far too complex to obtain a good quality system. Also using an iterative approach will allow users to see working versions of the software earlier. The following steps are performed during the System Design:

- Defining the initial architecture.
- Planning executable releases.
- Developing executable releases.

Defining The Initial Architecture

The internal structure or organization of a system is called *architecture*. When a system has a clean, well organized architecture it is easy to understand, test, maintain, and extend. Usually architectures are organized in layers. Each layer uses the services of the layers below and has no knowledge of the layers above. This facilitates changing each of these layers at any time without affecting the rest of the system (as long as the interface between them is maintained). There are two main tasks involved in defining the initial architecture:

- **Choosing major service software.** We have to decide which service software will be used during implementation. This software may include an operating system, a database manager, device interfaces, graphical user interfaces (GUIs), etc. This decision has to be made early in the design process as it will be a starting point for the remainder of the design decisions.
- **Defining class categories.** When new classes are added for implementation purposes, the system grows. New class categories may have to be created to contain these classes. It is important to keep the logical and physical aspects of the system independent of each other. This will facilitate the portability of the domain part of the system across multiple platforms and will also help in the reusability of the classes used only for implementation (e.g. the interface library).

Planning Executable Releases

As with design, trying to implement the whole system in one step is far too complex to obtain a good quality system. Instead, it is better to build the system in a series of executable releases. An *executable release* is a *mini-system* that performs a group of related functions and tasks.

An incremental plan to build the system using several executable releases has to be defined. Eventually these releases will be integrated to provide a complete system. One approach for planning executable releases is to try to reduce development risk. This means that each release should eliminate some risk in the project. Risk areas are those areas in the system's design or requirements that are possibly incomplete or incorrect. The customer may not be sure about a certain part of the system and will need to see the system working to make sure that that part of the system is what he/she really needs. Also new services, hardware or another systems that were not tried will be tested to see if they comply with what was expected of them or if there is a need to replace them.

The *executable release plan* should contain the goal of the release, the classes to be implemented, the use cases to be implemented and the required inputs and outputs. Testing will be done after each release to ensure that the goals of the executable release are met.

Developing Executable Releases

In order to develop an executable release the following tasks have to be performed:

- **Adding control classes.** Control classes model functionality that is not present in any of the classes of the system. They make different objects collaborate to provide certain behavior. Instances of control classes are usually temporal and only last during the execution of an activity.
- **Detailing the implementation of operations.** Many operations are sufficiently simple to be detailed during analysis. Other operations, in particular those that involve many objects, need further definition. New object diagrams will be developed for these operations. Also, for some complex operations, object diagrams might not be sufficient to illustrate the steps of the operation.

In these cases, an algorithm will provide a better definition of the operation. Pseudo code or the implementation language could be used to express this algorithm.

- **Implementing relationships.** There are two issues to consider: navigational paths and containment. During analysis, associations are defined between classes. These relationships are bi-directional, but some of them are traversed only in one direction. During design we add the navigation paths to the associations. We also have to define how all the relationships between classes will be implemented: by containing the object or by pointers or references. Also some container classes might need to be defined in order to implement *One-to-Many* and *Many-to-Many* relationships (e.g. the class **Set**).

- **Defining access control.** During analysis one could assume that every class has access to every attribute and operation of any other class. During design we try encapsulate each class so that the implementation is completely hidden from its public interface. This will help localize all the effects of change.

Deliverables of System Design

The deliverables of system design include:

- **Completed class diagrams** that show the new implementation classes and the categories added to the system.

- **Completed class specifications** which show the implementation details added such as algorithms to carry out operations, members added to the classes for internal operation and access control of the members.

- **Design object diagrams** for the non-trivial operations.

- **Executable release plans**

- **Architectural descriptions** which describe the choices of hardware and software for the system.

3.3 Summary

In this chapter the Booch methodology is introduced. This software development methodology provides a standardized means of presenting and communicating the system requirements and design decisions for the DIDP system. The Booch notation is explained focusing in the subset of the notation used in this project. Each of the three steps of the methodology (i.e. requirements analysis, domain analysis and system design) is discussed in detail. The deliverables for each step are also presented.

Chapter 4

Requirements Analysis

The first step in defining the requirements of the DIDP system is to understand the problem domain and what is expected from the system. In order to achieve this goal several activities are performed:

- **Reading the project proposal.** The first task was to know what the goal of the DIDP project was. The proposal provided a starting point in defining the scope of the system.
- **Observing the real system.** Seeing the environment and conditions in which the users of the system work provided a good idea about some of the characteristics that the system must comply. For example, it is not uncommon that paramedics make notes on small pieces of paper while treating the patient and they complete the formal report later once the patient is in the hospital. This means that the data is not always entered in the system in the same sequence as it happens in real life and thus the validation mechanisms should contemplate this situation.
- **Consulting with domain experts.** Many paramedics, doctors and nurses were interviewed during this process. It is very important to understand that each professional in the health system has a different need in order to build a system that satisfies all its users. For example, doctors use more free-format forms while nurses use charts in order to enter information.
- **Revising forms.** Many paper forms that are used in the hospital and in the ambulances were reviewed in an attempt to become more familiar with the

vocabulary and the type of data collected. Forms that are no longer used were also revised to understand why they were discarded.

4.1 Patient's Flow

In this section we explain the process that is followed when a patient is injured and 911 is called until the patient is discharged from the hospital and reintegrated into the society (see Figure 4.1).

Emergency Medical Service (EMS). Injuries can happen anywhere: at home, at work, on the road. Once 911 is called, an ambulance is dispatched to the scene of the injury. This ambulance could be an air or ground ambulance depending on the location of the incident. The emergency medical technicians and paramedics in the ambulance serve as an extension of the physicians in the field and work under medical direction. The difference between an emergency medical technician and a paramedic is that paramedics can perform more advanced procedures such as intubating the airway and giving medications. It is important that the injured person is treated in the first hour of being injured to have a better chance of survival. The *Emergency Medical Service (EMS)* has the responsibility of transporting the patient to a hospital as soon as possible and to stabilize the patient's condition on the way to the hospital.

Emergency Room (ER). Once the ambulance arrives at the hospital the patient is brought into the *Emergency Room (ER)*. Emergency departments have many types of professionals such as emergency doctors, nurses, specialists-orthopedics, respiratory/orthopedic/x-ray technicians, and students in these areas. An emergency doctor assesses the patient and if the patient has a only minor injury he/she is sent home after receiving the necessary treatment. In more severe cases the patient is referred to specialists within the hospital and could be transfered to the Operating Room (OR), the Intensive Care Unit (ICU) or to the Ward.

Operating Room (OR). Patients that need surgery will be brought to the *Operating Room (OR)*. Anaesthetists, surgeons and nurses are among the type of

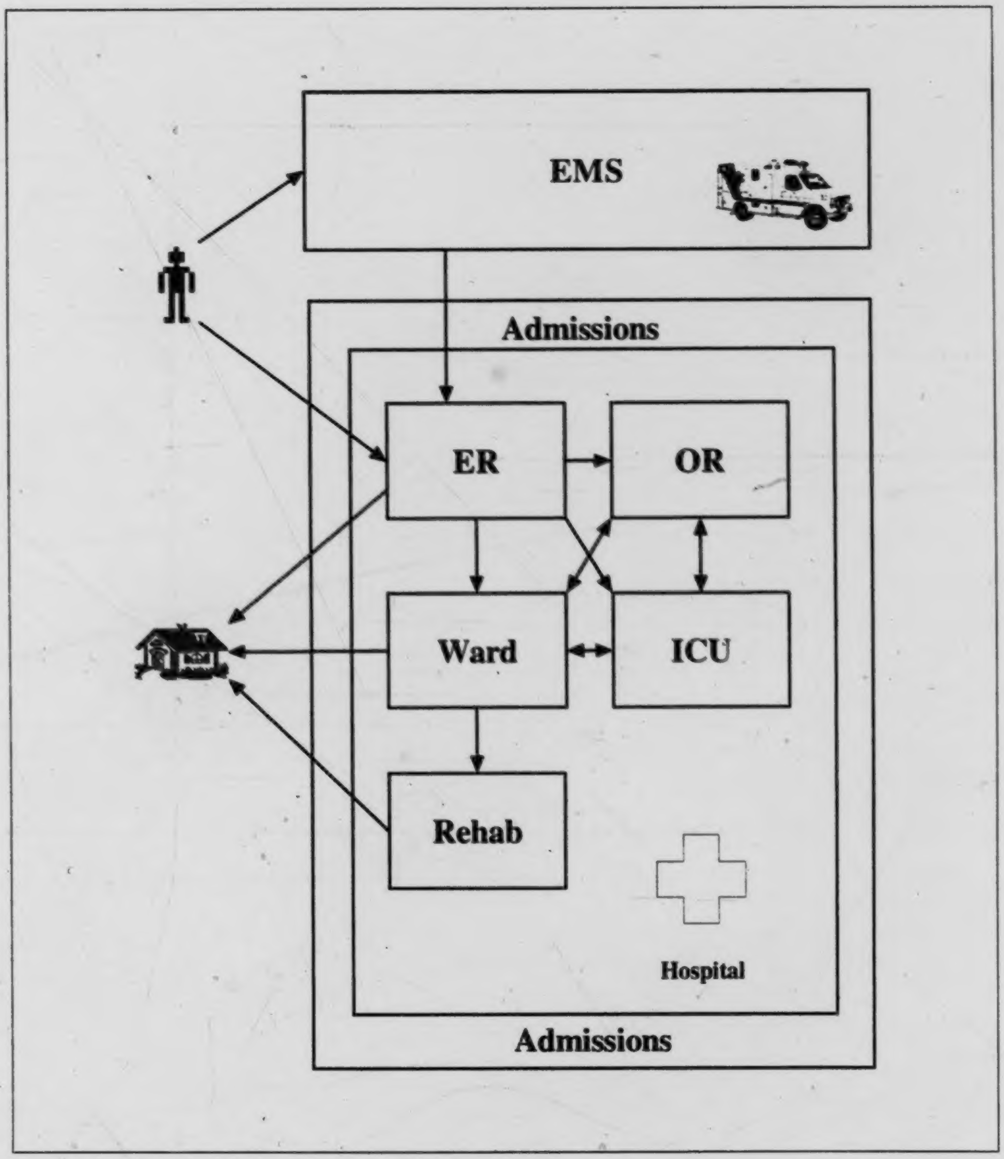
EMS
Admissions
ER
OR
Ward
ICU
Rehab
Hospital
Admissions

Figure 4.1: Patient's Flow

personnel we can find in this unit. Once the patient is operated he/she can be transfered to the ICU or the Ward depending on his/her condition. It could happen that a patient needs more than one surgery and is thus transfered back and forth from the ICU/Ward to the OR.

Intensive Care Unit (ICU). Patients that need stabilization and close observation are brought to the *Intensive Care Unit (ICU)*. Here special nurses and doctors take care of them until they are well enough to go to the Ward or die.

Ward. Patients are sent to the *Ward* when they are not well enough to go home or to rehabilitation but they do not need very close observations as to send them to the ICU. Nurses, technicians and doctors work here. They are capable of taking care of orthopedic and internal injuries. Once the patient is recovered he/she is discharged and sent home or is transfered to the Rehabilitation unit.

Rehabilitation. The objective of the *Rehabilitation* unit is to give back to the patients the skills they had before the injury. Occupational therapists, physiotherapists and speech therapists are part of a team devoted to maximize the skills an injured person can gain. Once a patient is rehabilitated he/she is sent home.

There are other units in the hospital such as *Laboratory Services*, *Radiology Department*, etc. that provide support to all the professionals working in the units mentioned above. Also the *Admissions Office* takes care of the administrative part of the hospital.

4.2 System Architecture

Figure 4.2 illustrates all the major elements in our architecture. We have one or more hand-held computers in each of the hospital units and in the ambulances. These units communicate with the database server through a network. A few details about this network should be mentioned. From the system's perspective the choice of a particular network is not important as long as it provides a reliable service. The presence of hand-held computers suggests that a wireless network could be used. As

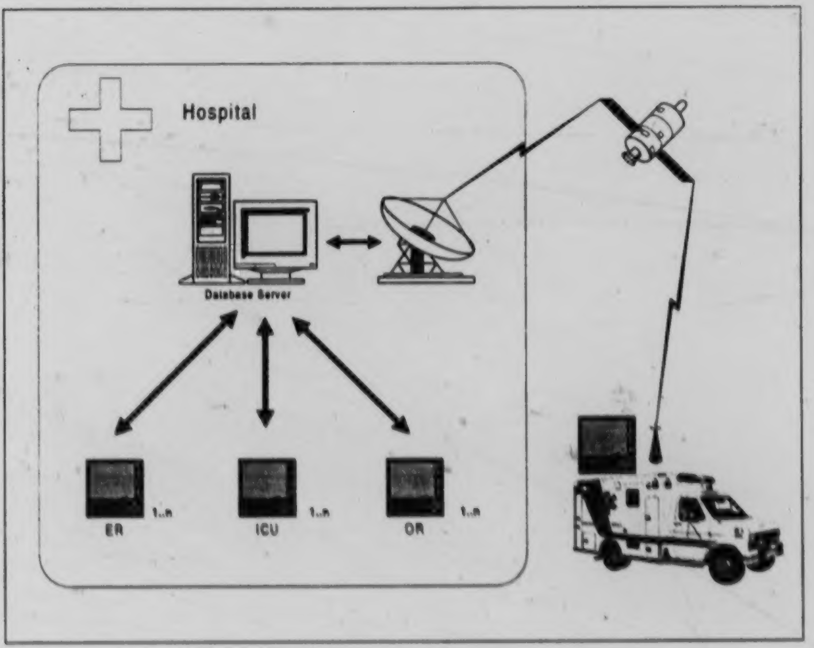

Figure 4.2: System Architecture

observed during analysis, the way of capturing data will be much easier using these units because of the nature of the activities performed by the users of the system. For example, there is usually no space in an operating room to place a desktop computer. If a wireless network is not used, the hand-held units could be used stand-alone and then the collected data downloaded to the database server with a certain frequency.

4.2.1 Data Processing Architecture

The data processing architecture of the system, including some components not yet developed, is depicted in Figure 4.3. The *data collection* component of the architecture utilizes pen-based hand-held computers to be employed by the emergency medical services and the hospital personnel to capture the most important patient-related information. The *database server* component of the system stores the collected data and allows sophisticated analysis of the data through a visual query interface. The *mapper* component will link the data collection and database server components so that data could be transfered between them.

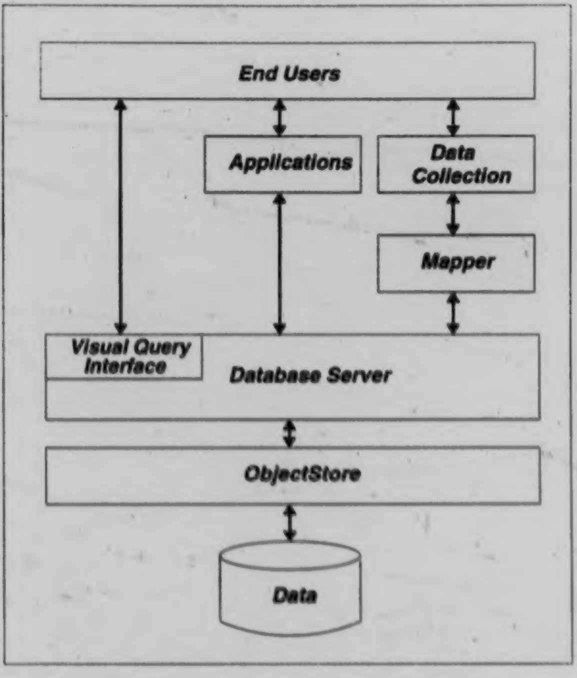

Figure 4.3: Data Processing Architecture

The object-oriented DBMS used to develop the database server is *ObjectStore* [Obj95]. ObjectStore has been used in many applications where high performance, reliability, concurrency and scalability are among the rigorous requirements to be met, and where conventional databases are unable to meet user requirements due to their limited ability to accommodate complex data and relationships [Obj97]. ObjectStore provides native support for extended data types such as image, free text, video, audio, time series, spatial and HTML objects, as well as for the extended relationships among non-tabular unstructed data. This enables the building of applications that would be difficult or impossible to implement with conventional relational or object-relational DBMSs.

This thesis deals with the database server layer of the architecture. The *data collection* component is being developed in the Department of Public Health Sciences at the University of Alberta. Future Work, as discussed in Chapter 7, includes the development of the visual query interface, the mapper, and several other applications to interact with the database server and provide services to the end users.

4.3 Database Requirements

After a careful analysis of all the data gathered in each of the different units of the hospital and the EMS we define the *minimum data set* for each of them. A *minimum data set* is composed of the least number of items of information which provide most of the data required by the majority of the users. For this phase of the project we focus on the information gathered in the EMS, ER, ICU, OR and the Admissions office.

There is significant commonality in the data requirements of each unit, even if the commonalities are exhibited at different levels of detail. The type of collected data can be classified into the following groups:

- **Patient identification and health information.** Includes demographics information about the patients, medications taken on a regular basis, medic alerts, allergies and other medical problems the patients have.
- **General information about the visits to the hospital/EMS.** Includes detailed chronological information about the visits of the patients to the different units, information needed by the Workers' Compensations Board and Social Services, and information on the incidents in which the patients were injured.
- **Medications, antibiotics and IVs.** Includes all the medications, antibiotics and IVs provided to the patients.
- **Diagnostic images and lab exams.** Includes the laboratory exams and images ordered for the patients including their results.
- **Invasive therapy, instrumentation and fluids.** Includes the genito-urinary procedures done to the patients, the instrumentation applied and the input/output fluid assessments.
- **Critical incidents and personnel.** Includes the critical incidents occurred to the patients and the personnel contacted during the patients' visits.
- **Gastrointestinal assessment.** Includes gastrointestinal exams, ostomy, stoma and stool assessments.

- **Central nervous system assessment.** Includes general central nervous system assessments, pain assessments, spinal precautions applied to the patients and intercranial probe readings.

- **Respiratory assessment.** Includes chest exams, respiration support devices applied, airway procedures done and ventilator control readings.

- **Vital Signs assessment.** Includes pulse, respiration, blood pressure, pupil, skin and Glasgow comma scale assessments.

- **Other assessments.** Includes injury and cardiovascular system assessments and musculoskeletal devices applied.

- **Specific information about OR anaesthesia and procedures.** Includes pre-assessments of the patients before the operations, anaesthesia setup information, operation procedures done and monitor readings during the operations.

- **Specific information about the EMS visit.** Includes general assessments and treatments done and information about the ambulance runs.

4.3.1 The Database Server Charter

In order to conclude the requirements analysis we have to define the scope and responsibilities of the database server. The key responsibilities of the database server include:

- To provide persistent (i.e. permanent) storage for the data collected from the scene of the injury to the time the patient is discharged and re-integrated into the society. In this first version of the server we consider the information gathered in the EMS, ER, ICU, OR and the Admissions office.

- To ensure the integrity of the data enforcing the corresponding integrity constraints.

- To provide the necessary recovery mechanisms in case of hardware or software failures.

- To provide a mechanism that keeps track of when a user performs an operation on the data.

- To provide a uniform interface for the application programmers to write applications that interact with the database server.

4.4 Summary

In this chapter the requirements analysis is presented. The activities performed to understand the problem domain, and the flow of a patient since he/she is injured to the time of discharge from the hospital and reintegration into the society are discussed. The system and data processing architecture is introduced. The database requirements and the scope and responsibilities of the database server are also explained.

Chapter 5

The Design

Database design involves the definition of a precise object-oriented model of the DIDP system and its environment. The DIDP database design was kept simple and flexible to accommodate future changes in information needs and operating conditions. The programming language and database chosen for implementation (C++/ObjectStore) also influenced the design.

In the following sections the motivations behind each design decision and the influence of the implementation on the design are discussed.

5.1 Design Tool - Rational Rose

Automated tools help free analysts and designers from some of the tedious tasks of modeling so that they can concentrate on the truly creative aspects of analysis and design. There are certain things that these tools can do and others that they can not. Automated tools can, for example, enforce conventions, help with consistency checking of the model, tell whether or not a certain state in a state transition diagram is reachable and take care of the data dictionary. On the other hand, an automated tool can not tell when a new class has to be defined or how to simplify certain structures; that needs human insight.

For the DIDP system *Rational Rose/C++* [Rose95] was chosen as the visual modeling tool. Rational Rose supports the Booch notation and it is specifically designed for C++ developers who need to keep their application model synchronized with the implementation [Rose97]. Although Rational Rose provides C++ code generation, this feature was not used in this project since it only provides support for mapping persistent objects to a relational DBMS, not to the object-oriented DBMS that was used in this project.

5.2 The Model

The overview of the DIDP model is depicted on Figure 5.1. For each of the functional groups identified during the requirements analysis there is one class category. There is also an extra category called *General Classes* that contains the classes that are used by all the categories in the model.

The classes contained in a category are private and can not be accessed from any other category. Then only exceptions are those classes listed in the category icons that are offered as public. If a category needs one of the classes offered as public by another category, a *use* relationship has to be specified between the two. For example, the classes contained in the *EMS Specific Information* category can access the classes **Next Of Kin** and **Patient** contained in the *Patient Identification and Health Information* category. The classes contained in the *General Classes* category can be accessed by all the classes in the model because the category is marked as *global*.

Each category in the diagram has a class diagram associated with it. We describe each of these diagrams in the following sections. The complete set of class specifications are listed in Appendix A.

5.2.1 General Classes

As mentioned before, the *General Classes* category contains the classes that are used by all the categories in the model (see Figure 5.2).

We assume the existence of certain basic data types such as **integer, positive (or unsigned) integer, real, boolean, date, time** and **string** that are not shown in the diagrams but are defined in the class specifications for completeness.

In order to provide a basic auditing mechanism, an abstract class called **Security** was defined. This is the root class of the model and every class inherits from it either directly or through its superclass. **Security** provides a method called *TimeStamp()* that updates the date/time and user id stored in the object that calls it. Objects can

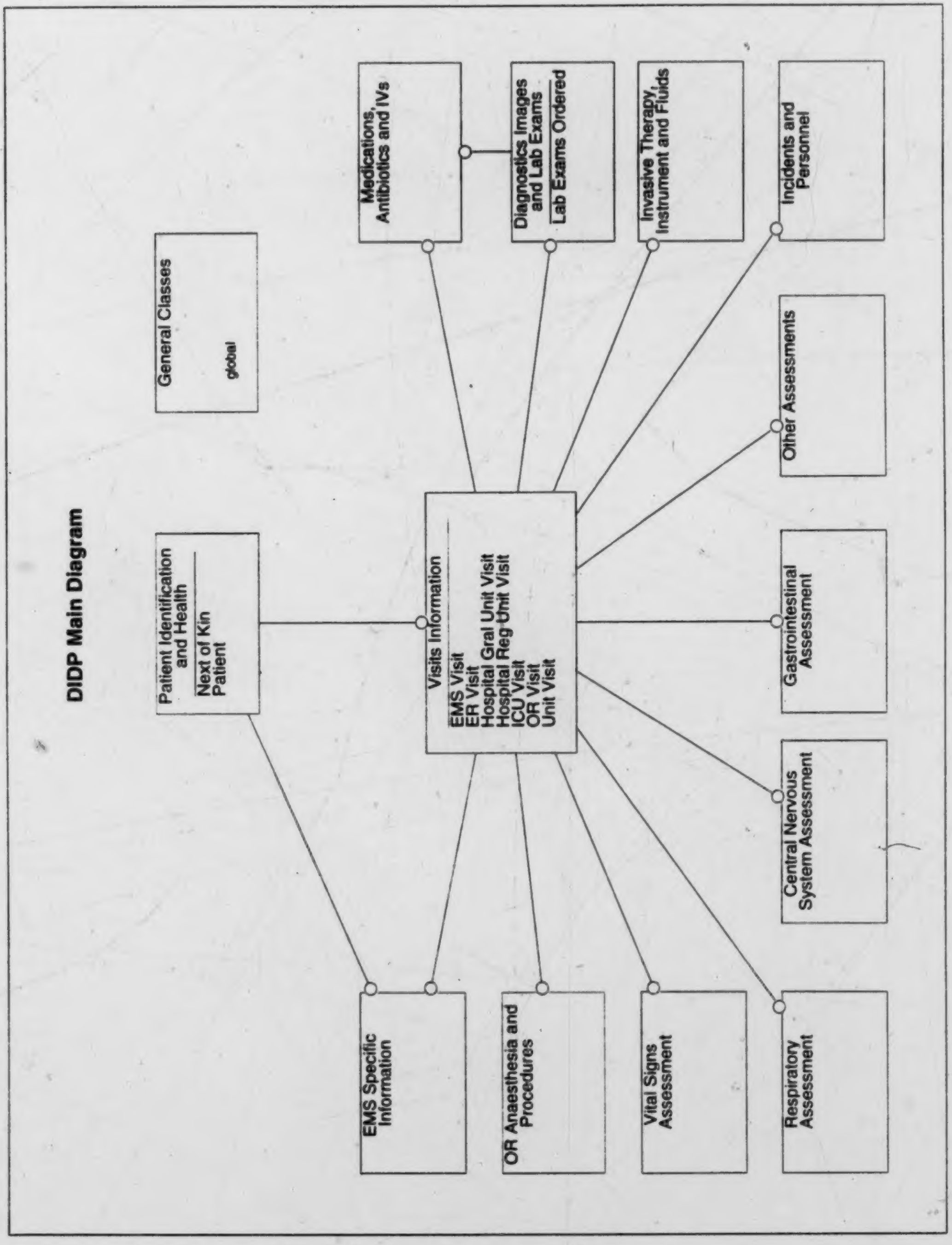
DIDP Main Diagram
Patient Identification and Health
Next of Kin
Patient
General Classes
global
Visits Information
EMS Visit
ER Visit
Hospital Gral Unit Visit
Hospital Reg Unit Visit
ICU Visit
OR Visit
Unit Visit
EMS Specific Information
OR Anaesthesia and Procedures
Vital Signs Assessment
Respiratory Assessment
Central Nervous System Assessment
Gastrointestinal Assessment
Other Assessments
Incidents and Personnel
Invasive Therapy, Instrument and Fluids
Diagnostics Images and Lab Exams
Lab Exams Ordered
Medications, Antibiotics and IVs

Figure 5.1: DIDP Main Diagram

invoke this method when they are created or each time they are being modified to record who created/modified the object and when. This mechanism has its limitations, as discussed in Chapter 7, but serves as a basic auditing tool that will help to develop a more extensive auditing mechanism in a future version of the system.

The abstract class **Named Object** provides the common attributes and behavior needed by all the classes that have *name* as an attribute. In particular, it simplifies the implementation of those subclasses of **Named Object** that do not have any extra attributes or behavior other than the ones inherited. There are many of these classes in the model where the only difference among them is their semantic definition. For example **Color** and **Drug Type** have exactly the same attributes and behavior, but they are used to model very different entities.

The abstract class **Descripted Object** works exactly like **Named Object** except that it provides the attributes and behavior needed by the classes that have *description* as an attribute instead of *name*.

The abstract class **Ranged Value** provides the common attributes and behavior for those classes that have a *name* and a *range* of valid values. For example, every laboratory exam has a name and a range of possible valid values for the result.

Because the inheritance relationships between the four classes discussed so far and their subclasses do not give any insight in the problem domain itself, they are not shown in the rest of the diagrams. For a detailed inheritance information see the class specifications in Appendix A.

The classes **Age**, **Pname** and **Address** are considered *extended types*. This means that the extents of these classes are not maintained, and that the objects of these classes only exists as part of other objects by physical containment instead of pointers or references. **Age** represents the age of a person in years and months. **Pname** represents a person name, including the first name, middle name, surname and title. **Address** represents any regular postal address including street number, street name, apartment number, postal code and city. It also stores the latitude and longitude coordinates of the location to be used by a Global Positioning System in the future (see Chapter 7).

An instance of the class **ICD9 Code** represents a particular *International Classification of Disease, 9th revision* code. These codes have widespread international

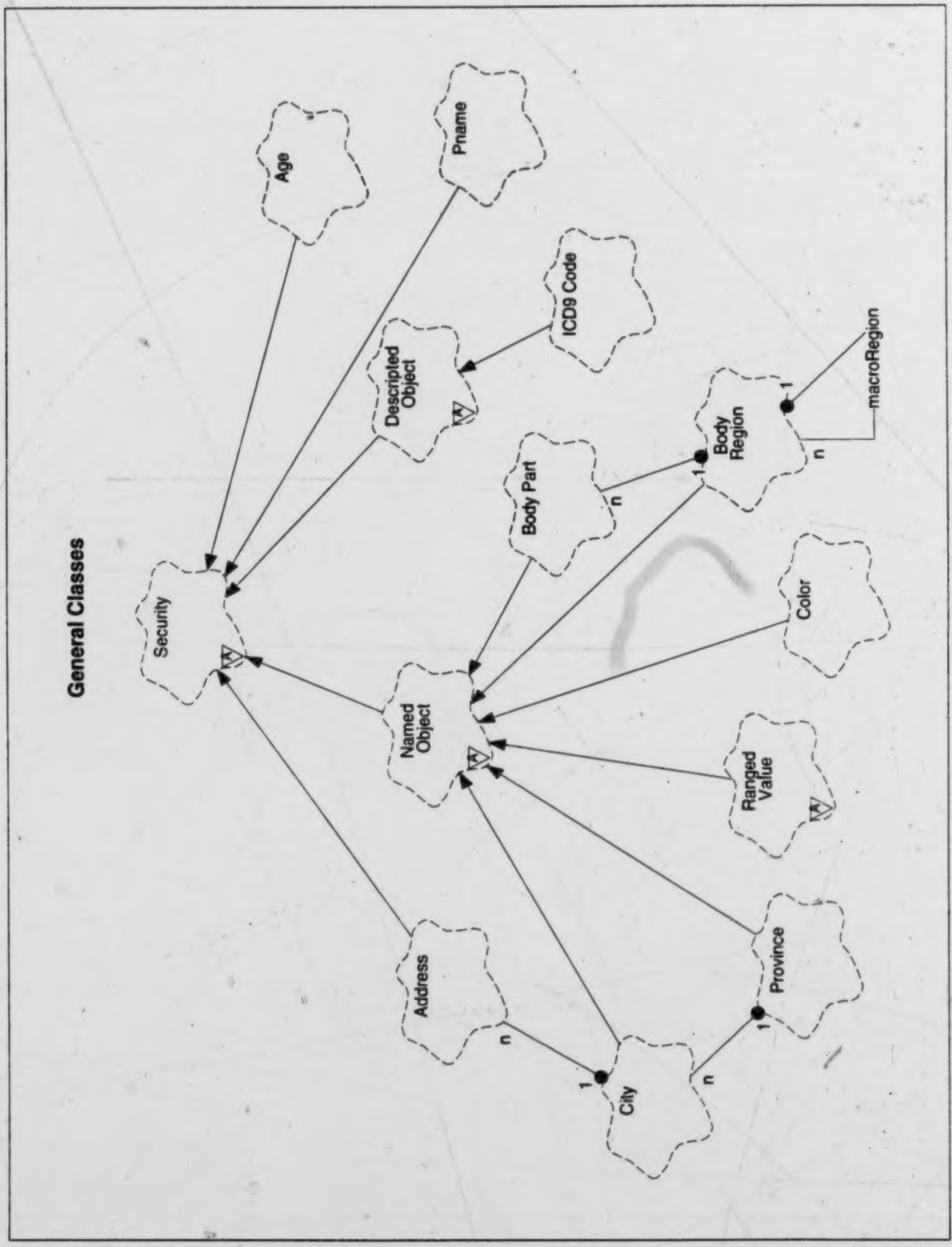
General Classes
Security
Age
Pname
Descripted Object
ICD9 Code
Body Part
Body Region
macroRegion
Named Object
Color
Ranged Value
Address
City
Province
n
1
n
1
1
n
n
1
n
1

Figure 5.2: General Classes

use to summarize anatomical diagnoses [Mac84] and are commonly used in discharge summary sheets.

We define a **Body Region** as any external region of the human body (e.g. chest, leg, head, etc.). A body region can have a macro region. For example the macro region for *hand* is *arm*. A body region has one or more body parts. A **Body Part** represents any part of the human body. It could be the skin of a body region, a joint, a bone, a blood vessel, a nerve, a muscle, a tendon, a ligament or an internal organ.

The class **Color** represents any color that a substance may have. The class **Province** and **City** are used to represent provinces and cities used throughout the model.

5.2.2 Patient Identification and Health Information

The *Patient Identification and Health Information* category contains the classes used to identify the patients and to maintain their health history (see Figure 5.3).

The abstract class **Person** includes information such as the name of the person, the home address and home telephone number. A **Patient** is any person that visits the hospital/EMS seeking attention. Each patient is identified by a patient identification number which is unique for the individual and is maintained through the different visits of the patient the hospital/EMS more than once. Other information about the patient includes the gender, date of birth, health care number, blue cross number, and the name of the family physician. If the patient is a native, the treaty number and the **Band** he/she belongs to has to be identified. If a patient dies the date/time and cause of death are also recorded. A **Next Of Kin** has to be specified for each patient when possible. The active next of kin is the one with the latest date of assessment. The system maintains the previous nexts of kin for historical purposes.

For every health condition that a patient has, the system records the date/time in which the condition was first assessed and the date/time since when the condition is no longer valid. This is necessary to know what the hospital/EMS personnel knew about a patient at a certain point in time and to maintain a history of the patient's health conditions. The abstract class **Medical Condition** provides the attributes and behavior necessary to comply with this requirement.

The classes **Medic Alert**, **Health Problem** and **Allergy** represent all the medic

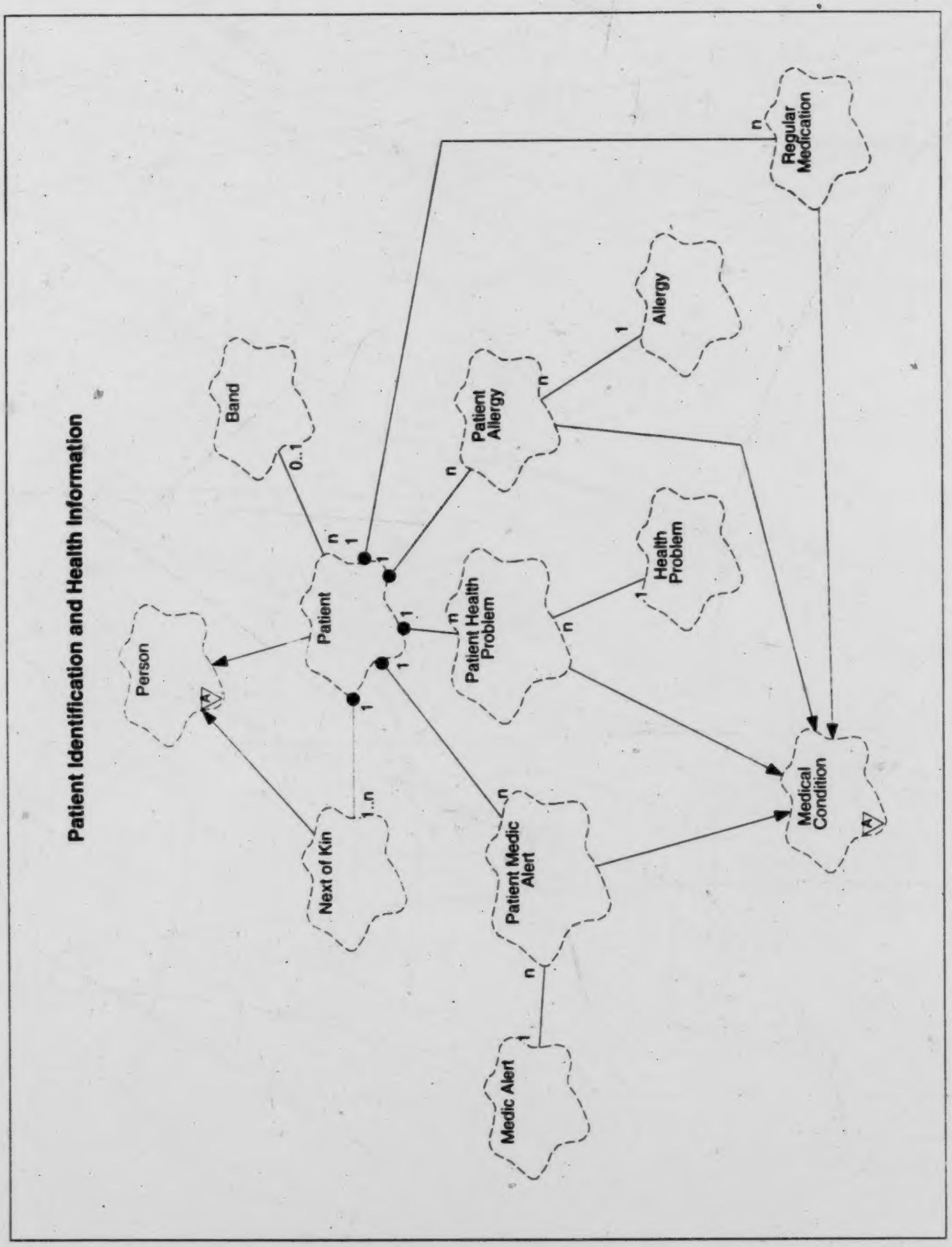
Patient Identification and Health Information
Person
Patient
Band
0..1
Next of Kin
1..n
Patient Allergy
Allergy
Patient Health Problem
Health Problem
Patient Medic Alert
Medic Alert
Regular Medication
Medical Condition
1
n

Figure 5.3: Patient Identification and Health Information

alerts, health problems and allergies a patient might have. The classes **Patient Medic Alert**, **Patient Health Problem** and **Patient Allergy** represent the medic alerts, health problems and allergies the patients have/had.

Each object of the class **Regular Medication** represents a medication that a patient takes on a regular basis. The information stored includes the name of the medication, the dose and frequency in which it is taken.

5.2.3 Visits Information

The *Visits Information* category contains the classes used to keep all the general information related to the patients' visits to the hospital/EMS (see Figure 5.4).

Every patient can visit the hospital more than once. A **Patient Visit** represents a particular visit of a patient. Within the hospital, the patient can visit many units (i.e. ER, ICU, OR). The design considers EMS as one of these units since data collection is very similar in all of them. The information related to a visit includes the weight and height of the patient, the date/time the patient arrived/left the hospital and the date/time of admission/discharge if the patient was admitted.

If a patient is injured while working, certain information is recorded for the Workers' Compensation Board (WCB). A **WCB Claim** keeps information about the patient's occupation, his/her social insurance number and the name of current employer. If the patient is under social assistance, some information is recorded for Social Services. **Social Service Info** keeps information such as the number of social service and the name of the social worker that is related to the patient.

A **Valuable** represents any valuable that a patient might have when he/she arrives at the hospital (e.g. a ring, a watch, a wallet, etc.). Each instance of **Patient Valuables** represents a collection of all the **Valuable**s that a patient had at check-in including the amount of cash (if any) and the name of the person who kept these valuables if they were not left with the patient.

Something very important to record for each patient visit is the information regarding the incident in which the patient was injured. This information is kept in **Incident Info**. The information recorded includes: whether the incident was indoors or outdoors, all the **Safety Devices** that the patient was using during the incident and the external **Cause**s of injury. Each instance of **Cause** represents a different

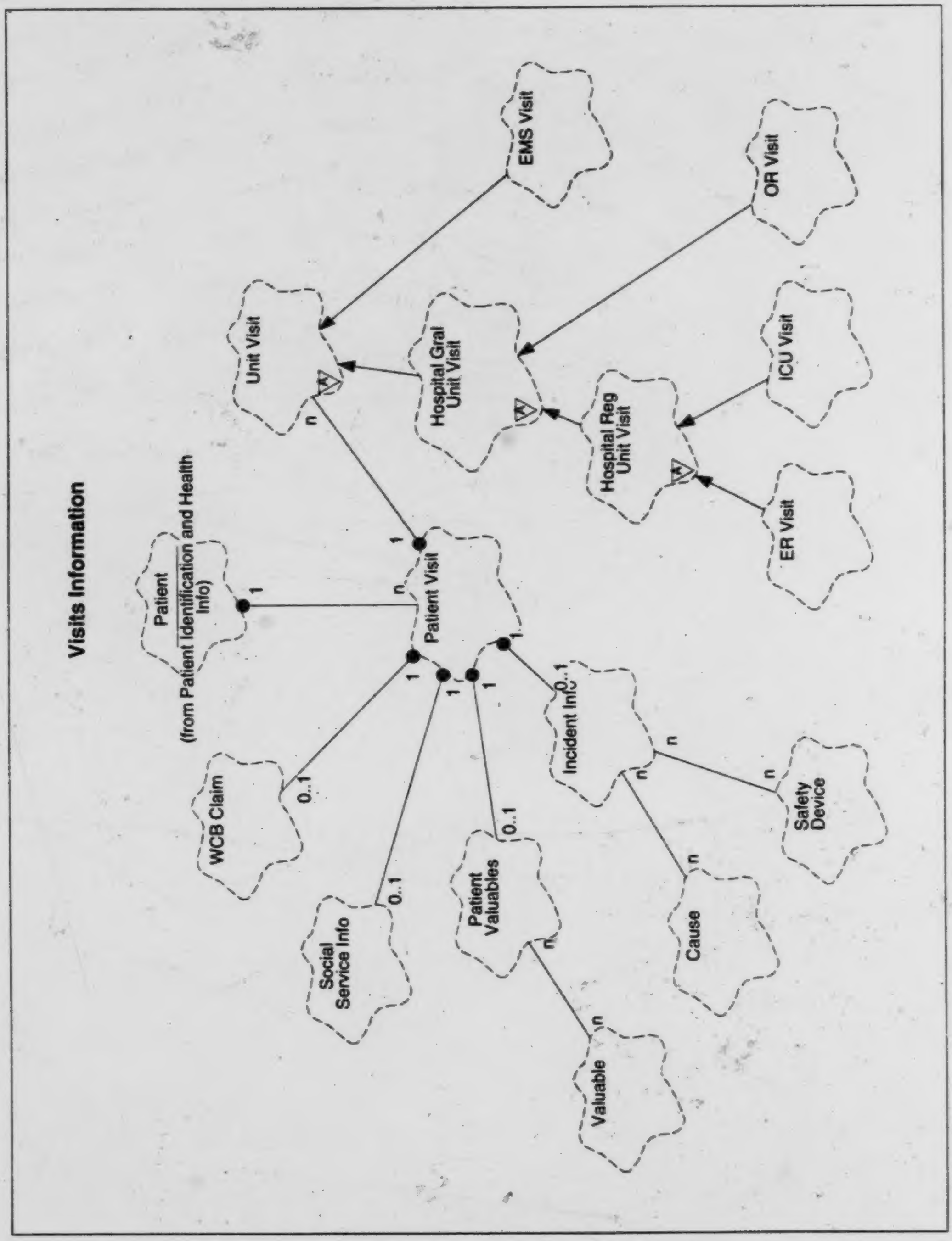
Visits Information
Patient
(from Patient Identification and Health Info)
Patient Visit
Unit Visit
EMS Visit
Hospital Gral Unit Visit
OR Visit
Hospital Reg Unit Visit
ICU Visit
ER Visit
Incident Info
Safety Device
Cause
WCB Claim
Social Service Info
Patient Valuables
Valuable
1
n
0..1

Figure 5.4: Visits Information

E-code. The E-codes are a subset of the ICD9 codes that are used for codifying the external causes of injury. Ribbeck et al [RRTB92] have first shown that E-coding is a valuable method for injury surveillance that can be easily performed in Emergency Departments, and that its value is essential for injury prevention research.

In different units, different information is recorded. In a first draft of the design one class was created per possible unit visit: **ER visit, ICU Visit, OR Visit** and **EMS Visit**. However, there are many commonalities among them which suggests the creation of superclasses. An instance of the abstract class **Unit Visit** represents a visit to *any unit*. There are certain activities such as giving drugs or inserting IVs that could be performed in every unit. An instance of the abstract class **Hospital Gral Unit Visit** represents a visit to any *hospital unit*. Certain things such as diagnostic images or lab exams are done in the hospital and can not be done in the EMS. An instance of the abstract class **Hospital Reg Unit Visit** represents a visit to *any unit in the hospital that is not OR*. OR has particular data requirements that are different from the rest of the hospital units.

5.2.4 Medications, Antibiotics and IVs

The *Medications, Antibiotics and IVs* category contains the classes used to keep information about the medications, antibiotics and IVs given to the patients during their visits (see Figure 5.5).

The class **Drug** represents all the drugs that can be given to a patient. The information stored includes: the name of the drug, its **Drug Type**, the recommended dose per kilogram and any other relevant information about it.

The abstract class **Drug Given** represents any drug that was given to a patient. The information recorded includes the given **Drug**, the **Drug Route** by which it was administered, the dose, the date/time that started and ended, and the schedule on which it was given, if any. A **Medication Given** represents any drug given to the patient that is *not* an antibiotic. As we can see in the figure, medications can be administered in any unit, but antibiotics can only be administered in the hospital units. An **Antibiotic Given** represents any antibiotic administered to the patient. Sometimes, in order to decide which antibiotic should be given to a patient, a laboratory exam is ordered. The system keeps track of the relationship between the

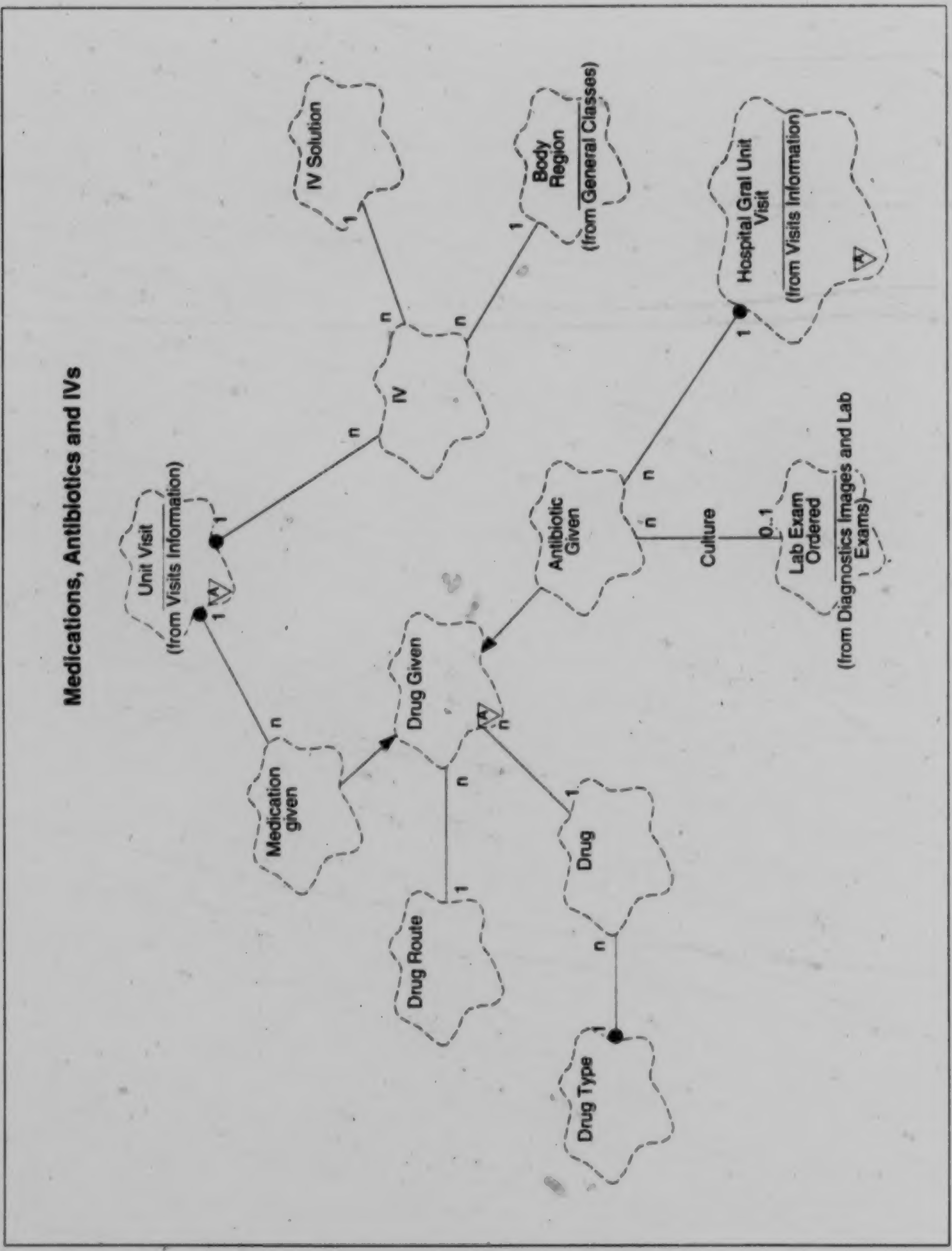
Medications, Antibiotics and IVs
Unit Visit
(from Visits Information)
IV Solution
Body Region
(from General Classes)
IV
Hospital Gral Unit Visit
(from Visits Information)
Antibiotic Given
Culture
0..1
Lab Exam Ordered
(from Diagnostics Images and Lab Exams)
Drug Given
Medication given
Drug Route
Drug
Drug Type
n
1

Figure 5.5: Medications, Antibiotics and IVs

Antibiotics Given and the **Lab Exams** (cultures) associated with them.

The information recorded for each **IV** done to a patient includes: the date/time it started and ended, the **IV Solution** given, the **Body Region** where it was inserted, the size of the needle used, and the rate infused in mm/hour.

5.2.5 Diagnostic Images and Lab Exams

The *Diagnostic Images and Lab Exams* category contains the classes used to keep information regarding the diagnostic images and laboratory exams ordered for the patients during their visits (see Figure 5.6).

Every **Image Ordered** for a patient is kept in the system. The information recorded includes: the type of **Image** ordered, the **Body Part** affected, the date/time it was ordered, the date/time it was done, and a textual description of the results. In the future the actual image will be stored together with this information (see Chapter 7).

A **Lab Exam** represents a laboratory exam that could be ordered for a patient. Every laboratory exam has a name and a range of possible values for its result. The system groups these laboratory exams into different **Lab Exam Types**. For every **Lab Exam Ordered** for a patient the type of **Lab Exam**, the date/time when it was ordered, the date/time the sample was taken (if any), the results, and the date/time these results were available are stored.

5.2.6 Invasive Therapy, Instrumentation and Fluids

The *Invasive Therapy, Instrumentation and Fluids* category contains the classes used to keep information regarding the genito-urinary procedures that are performed, the instrumentation applied, and the input/output fluid assessments of the patient (see Figure 5.7).

A **GU Procedure** represents a genito-urinary procedure that could be done to a patient in any regular hospital unit (i.e. ER and ICU). For each **GU Procedure Done** to a patient the type of **GU Procedure** and the date/time it started and ended, is stored. An **Instrument** represents any instrument that could be applied to a patient. For each **Instrument Applied** to a patient the type of **Instrument**, the

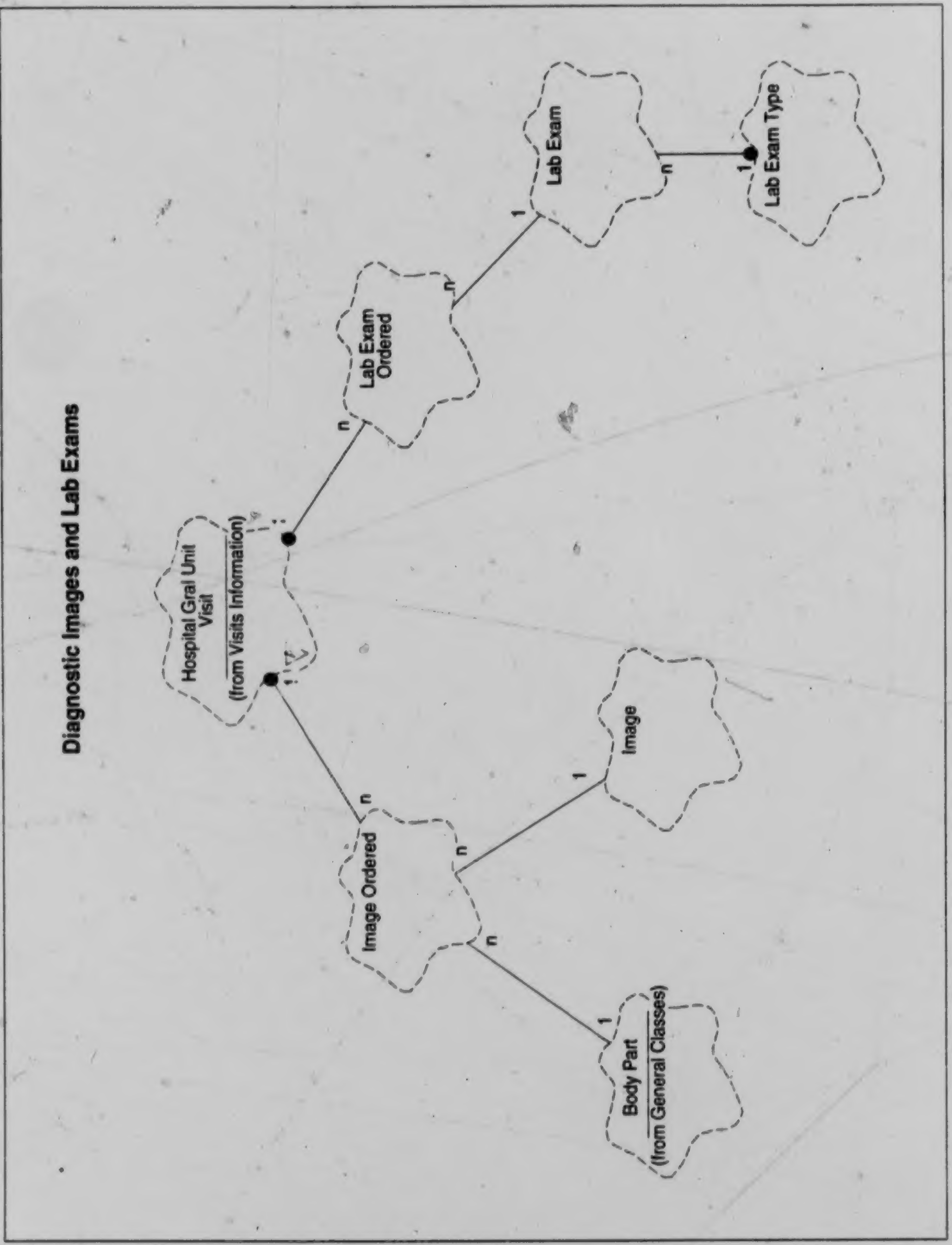
Diagnostic Images and Lab Exams
Hospital Gral Unit Visit
(from Visits Information)
Lab Exam Ordered
Lab Exam
Lab Exam Type
Image Ordered
Image
Body Part
(from General Classes)
1
n
1
n
n
1
n
1
n
n
1
n
n
1

Figure 5.6: Diagnostic Images and Lab Exams

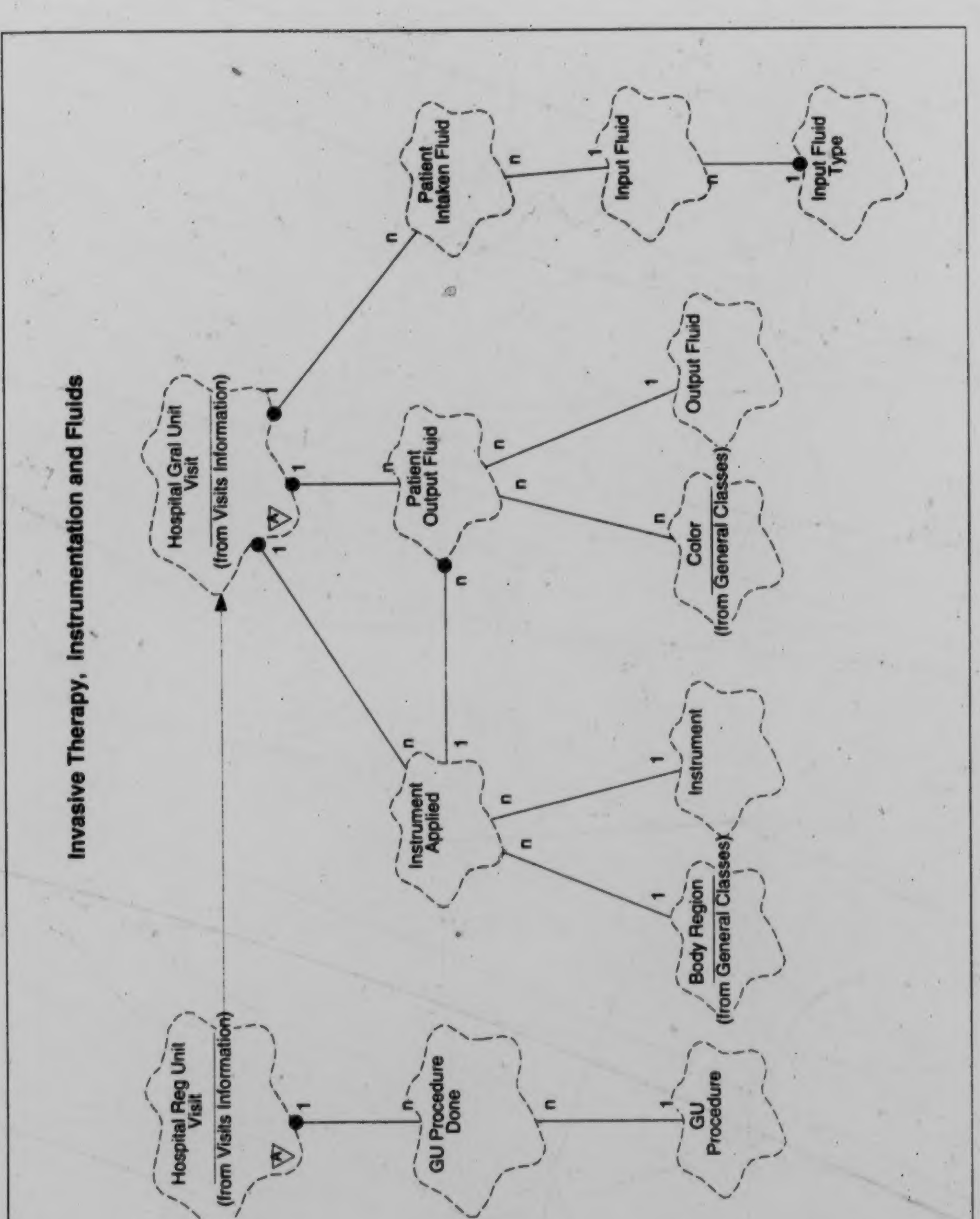
Invasive Therapy, Instrumentation and Fluids
Hospital Reg Unit Visit
(from Visits Information)
GU Procedure Done
GU Procedure
Hospital Gral Unit Visit
(from Visits Information)
Instrument Applied
Instrument
Body Region
(from General Classes)
Patient Output Fluid
Output Fluid
Color
(from General Classes)
Patient Intaken Fluid
Input Fluid
Input Fluid Type
1
n
n
1
1
n
n
1
n
1
n
1
1
n
1
n
n
n
1
n
1
n
1

Figure 5.7: Invasive Therapy, Instrumentation and Fluids

date/time of application, the **Body Region** where it was applied and the date/time it was removed is recorded.

An **Input Fluid** represents any fluid that could be given to a patient. These fluids are grouped into different **Input Fluid Types**. Each time a fluid is given to a patient, an instance of **Patient Intaken Fluid** is created and the type of **Input Fluid**, the date/time of assessment and amount of fluid is recorded.

An **Output Fluid** is any fluid that could come out of a patient's body. Each time a fluid comes out of a patient, an instance of **Patient Output Fluid** is created. The information recorded includes: the type of **Output Fluid**, the date/time of assessment and the amount, consistency and **Color** of the fluid. If the fluid comes out from a particular instrument that was applied to the patient, the relationship between the **Patient Output Fluid** and the **Instrument Applied** is recorded.

5.2.7 Incidents and Personnel

The *Incidents and Personnel* category contains the classes used to keep information regarding the critical incidents occurred to patients and the personnel contacted during the patients' visits (see Figure 5.8).

Every **Personnel Contacted** during a patient's visits is recorded by the system. The information recorded includes: the **Personnel Type** contacted, the name of the personnel, the date/time he/she was called, and the date/time he/she made the contact.

For every **Critical Incident Occurred** to a patient, the date/time of the incident, the **Critical Incident** and the **Critical Incident Reason** is recorded.

As depicted in Figure 5.8, every **Critical Incident** has a set of possible **Critical Incident Reasons**. What the figure does not show, however, is that only a **Critical Incident Reason** that belongs to the set of possible *reasons* for a **Critical Incident** can be chosen as the *incident reason* for a **Critical Incident Occurred**.

This problem is solved in the Booch notation by creating a dashed line between the **Critical Incident Occurred** and the *reasons* relationship and eliminating the *incident reason* and the *incident* relationships. The dashed line means that **Critical Incident Occurred** is associated with a pair of **Critical Incident-Critical Incident Reason**. We found some problems with this notation. If the model had an *m-n*

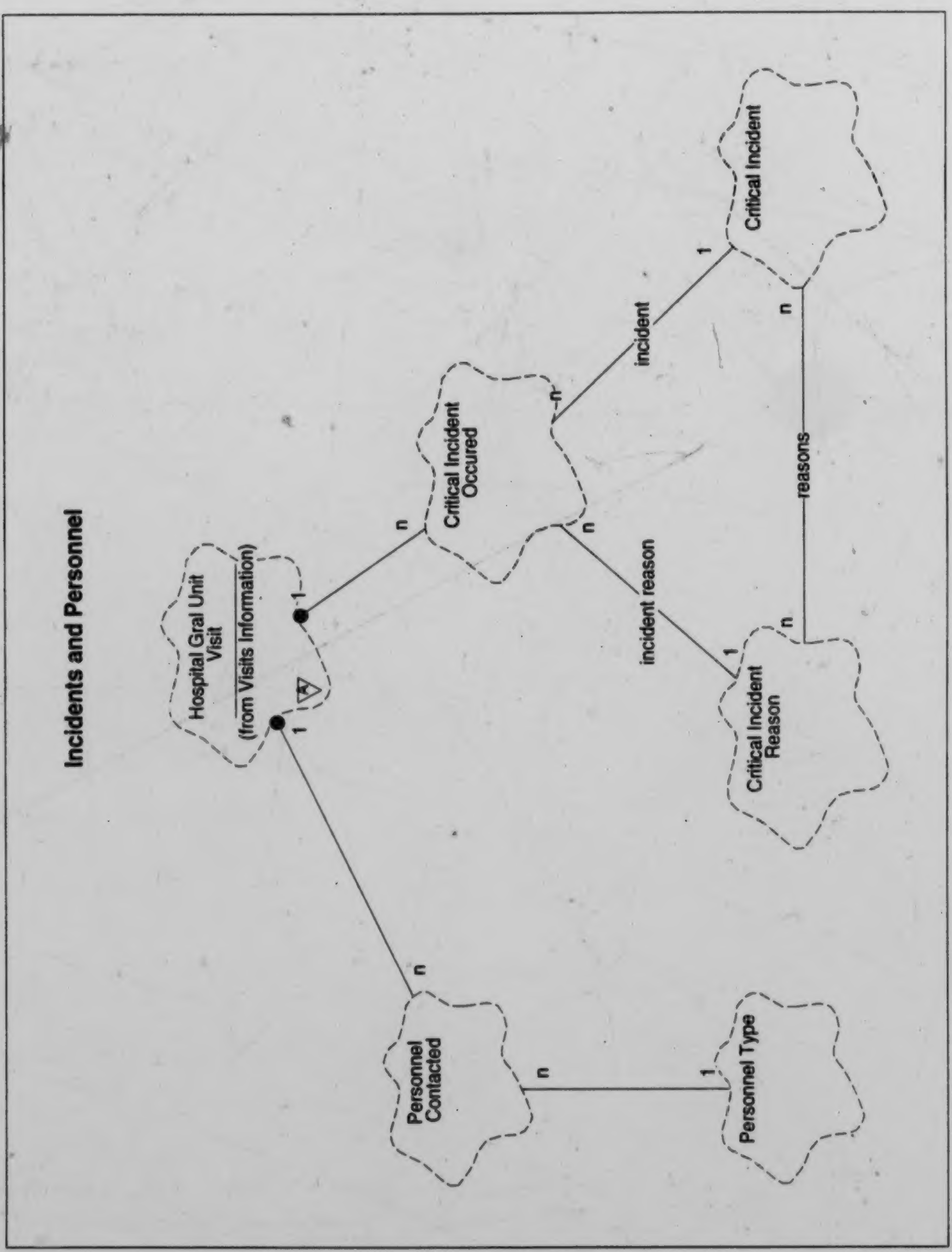
Incidents and Personnel
Hospital Gral Unit Visit
(from Visits Information)
Critical Incident Occured
Critical Incident
Critical Incident Reason
Personnel Contacted
Personnel Type
incident
incident reason
reasons
1
n
1
n
n
1
1
n
n
1
n
1

Figure 5.8: Incidents and Personnel

relationship between **Critical Incident Occurred** and **Critical Incident Reason**, using this notation would imply that we can have many **Critical Incident-Critical Incident Reason** pairs associated to a unique **Critical Incident Occurred** without indicating that the **Critical Incident** must be the same in all the pairs. Another drawback is that it could happen that we do not want to specify a reason every time an incident occurs, and this notation enforces to take at least one reason.

One way of solving this problem is to create an extra class that contains all the valid pairs of **Critical Incident - Critical Incident Reason** and creating a relationship between **Critical Incident Occurred** and this new class (see figure 5.9).

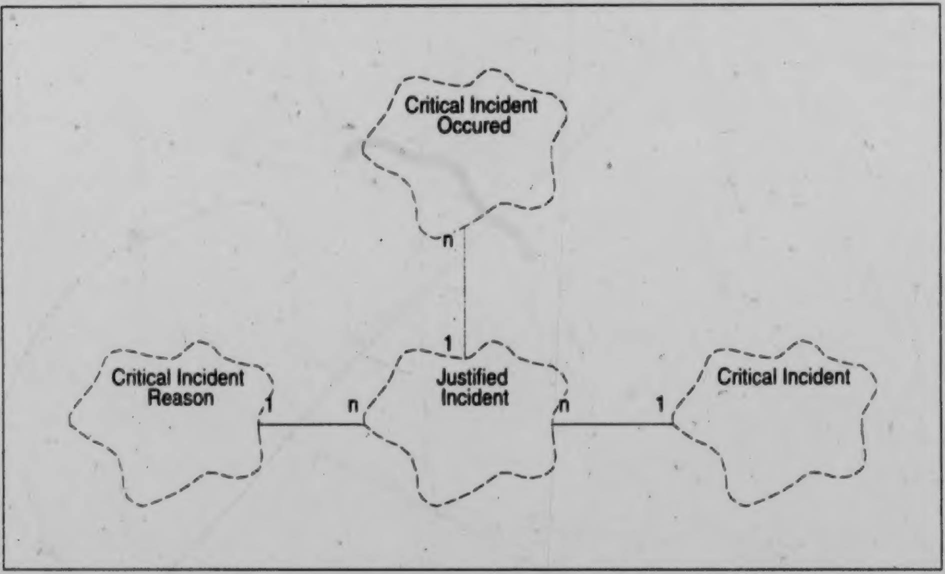

Figure 5.9: A possible solution for the critical incident problem

This would be good solution if each class were a flat structure, as in the relational model, because any *n-m* relationship would have to be implemented as a separate flat structure. With complex structures (i.e. objects), an *n-m* relationship can be implemented using sets without the necessity of any extra structures. Thus, inserting a new class is not a very good approach. This would force the model to have things that do not exist in the real world and also would make the implementation more complex. Furthermore, the *n-m* case mentioned before would not be solved by this method, and neither the enforcement of at least one reason.

What is needed to solve this problem is to show in the diagram the restriction of the *incident reason* relationship. For this purpose, we propose a new element in the notation: A dotted arrow between the relationship that has a restriction (i.e. *incident reason*) and the restriction relationship (i.e. *reasons*). The head of the arrow will point towards the restriction relationship. This would be read as: a **Critical Incident Occurred** can be associated with a **Critical Incident Reason** if and only if that reason belongs to the set of reasons valid for the **Critical Incident** associated with the **Critical Incident Occurred**. This solution works for *n-m relationships* and does not enforce the model to choose a reason if it is not needed. This notation is not presented in the diagram because the diagrams were generated using Rational Rose which does not support this notation or allows free hand drawing.

5.2.8 Other Assessments

The *Other Assessments* category contains the classes used to keep information regarding injury, musculoskeletal and central vascular system assessments done to the patients during their visits (see Figure 5.10).

For every **Injury** a patient presents, the date/time of the assessment, the affected **Body Part** and the **ICD9 Code** that applies have to be recorded.

A **Musskel Device** represents a musculoskeletal device that could be applied to a patient. A **Musskel Assessment** represents a musculoskeletal device that is applied to a patient. The type of **Musskel Device**, the **Body Region** where it is applied and the date/time of application and removal have to be recorded.

A **CVS Assessment** represents a cardiovascular system assessment done to a patient in any regular hospital unit. For every **CVS Assessment**, the date/time of assessment and the juglar venous pressure (JVP) have to be specified. Each time a **CVS Assessment** is done, a **Heart Assessment** and a **CVS Pulse Assessment** for each side of the body are also done. A **Heart Assessment** stores the different type of heart sounds found during the patient's heart assessment and the best way to apply the monitor leads in order to obtain the best configuration of the electrocardiogram (ECG) pattern. A **CVS Pulse Assessment** specifies which side of the body is being assessed and if a pulse is present or absent in different locations of the body.

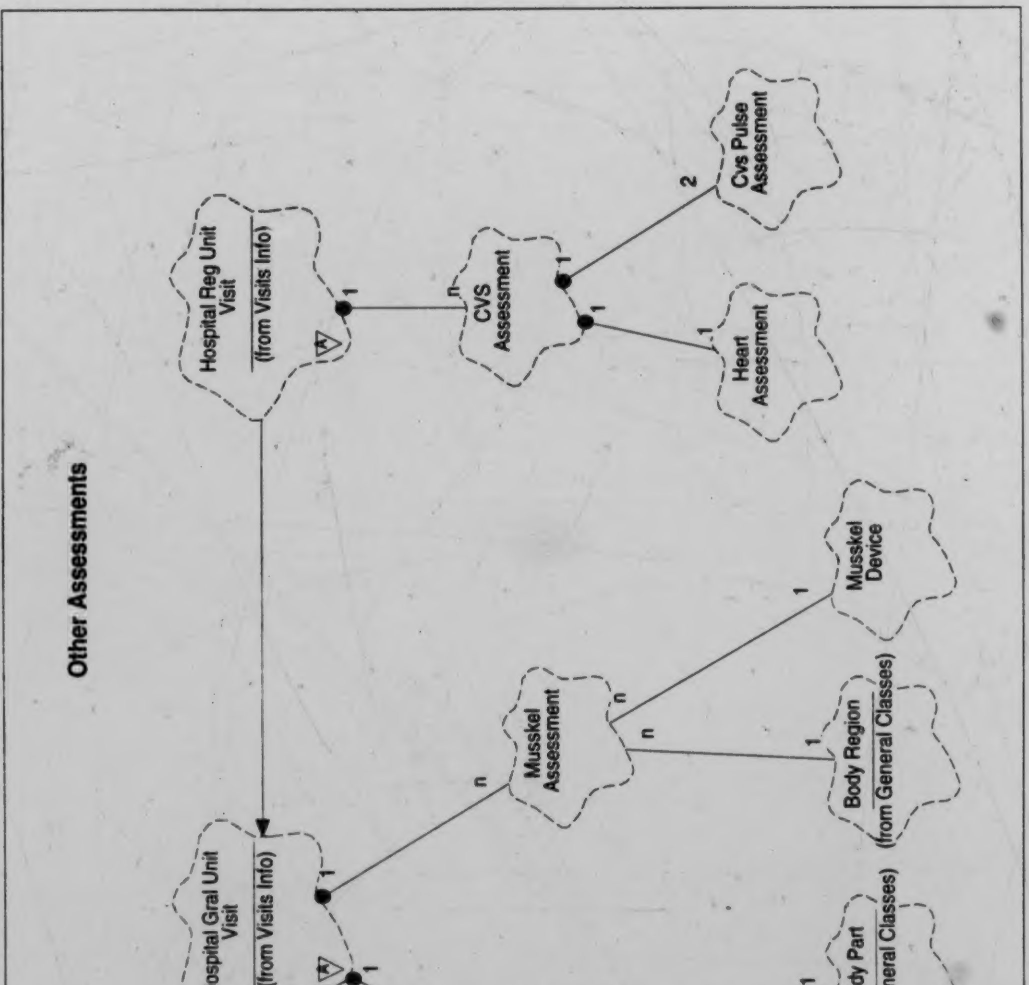
Other Assessments
Hospital Reg Unit Visit
(from Visits Info)
1
n
CVS Assessment
1
1
2
Cvs Pulse Assessment
1
Heart Assessment
Hospital Gral Unit Visit
(from Visits Info)
1
1
n
Musskel Assessment
n
n
1
Musskel Device
1
Body Region
(from General Classes)
n
Injury
n
n
1
Body Part
(from General Classes)
1
ICD9 Code
(from General Classes)

Figure 5.10: Other Assessments

5.2.9 Gastrointestinal Assessment

The *Gastrointestinal Assessment* category contains the classes used to keep information regarding the gastrointestinal exams, and the stool, ostomy and stoma assessments done to the patients during their visits (see Figure 5.11).

A **Gi Exam** represents a gastrointestinal exam done to a patient in any regular hospital unit. For every **Gi Exam** the following information has to be specified: the date/time of assessment, the shape of the abdomen, the bowel sounds, if the patient is nauseated or has cramps, and the results of the peritoneal lavage and rectal examination if they were done. Also, for each abdomen quadrant, an assessment indicating the rigidness, distension and tenderness of the zone has to be done. Each instance of **Gi Quadrant Assessment** represents the assessment of an abdomen quadrant for a patient.

The stool, ostomy and stoma assessments are only done in the ICU. For each **Stool Assessment** the date/time of assessment and the **Color** and consistency of the stool have to be recorded. For each ostomy done to a patient an **Ostomy Assessment** is recorded. The information regarding an ostomy includes: the date/time of assessment, the type of ostomy and the abdomen quadrant where it was done. If a mocous fistula was also done, the abdomen quadrant where it is located also has to be specified. For a **Stoma Assessment** the date/time of assessment, the integrity and the associated ostomy have to be recorded.

5.2.10 Central Nervous System Assessment

The *Central Nervous System Assessment* category contains the classes used to keep information regarding the general central nervous system (CNS) and pain assessments, the intercranial probe control and the spinal precautions applied to the patients during their visits (see Figure 5.12).

For each **Pain Assessment** done in any regular hospital unit, the date/time of assessment, the **Body Region** assessed, the severity, intensity and description of the pain are recorded.

A **CNS Assessment** represents a central nervous system general assessment done to a patient in any regular hospital unit. For every **CNS Assessment**, the

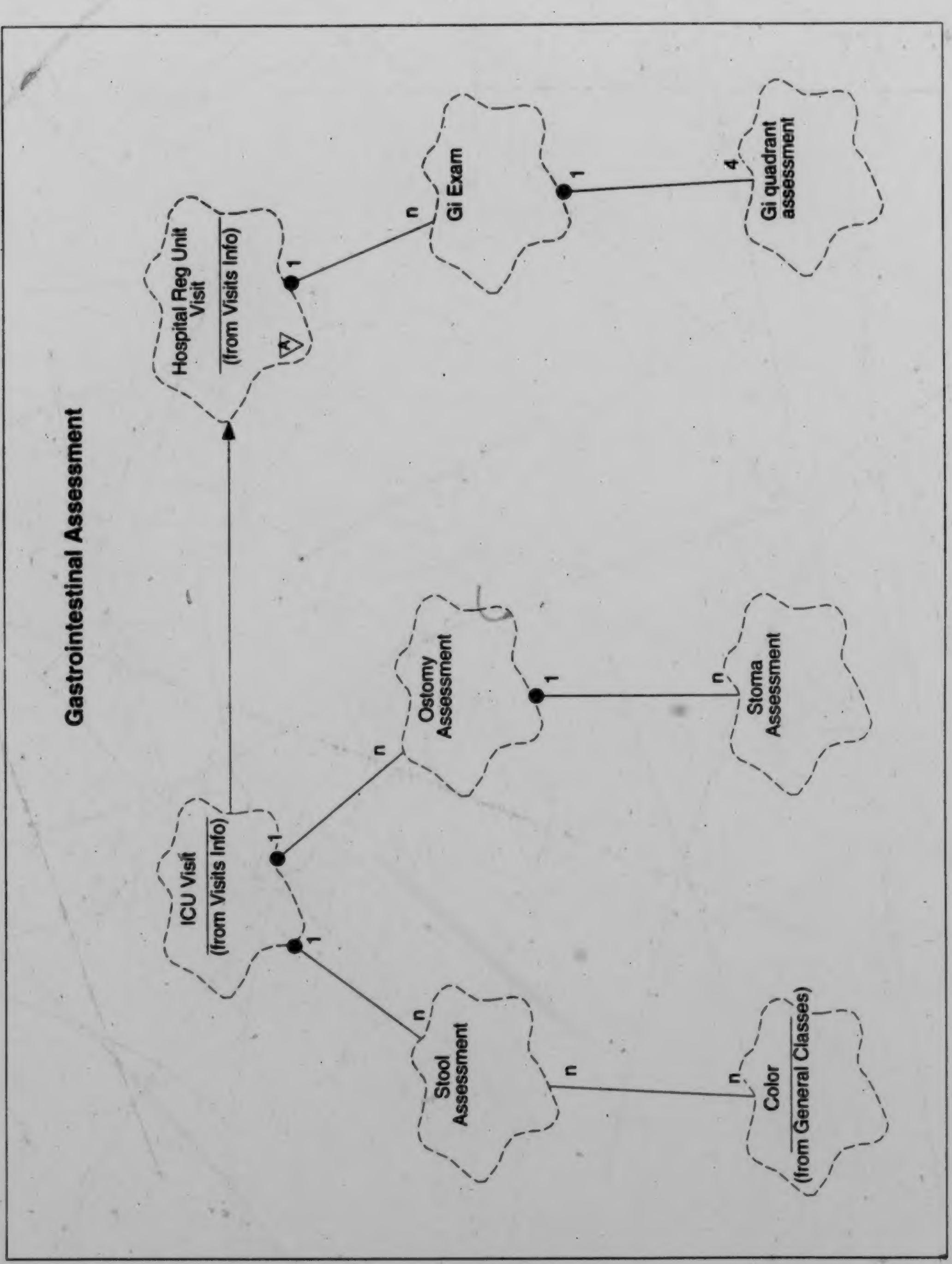
Gastrointestinal Assessment
Hospital Reg Unit Visit
(from Visits Info)
1
n
Gi Exam
1
4
Gi quadrant assessment
ICU Visit
(from Visits Info)
1
n
Ostomy Assessment
1
n
Stoma Assessment
1
n
Stool Assessment
n
n
Color
(from General Classes)

Figure 5.11: Gastrointestinal Assessment

date/time of assessment has to be specified. Each time a **CNS Assessment** is done, the **Reflexes** and **Movements** for each side of the patient's body are assessed. An instance of **Reflexes** specifies which side of the body is being assessed and if the reflexes are normal, absent or brisk for different parts of the body. An instance of **Movements** specifies which side of the body is being assessed and characterizes the movement of the arm and leg for that side.

The intercranial probe (ICP) control is only performed in the ICU. Each instance of **ICP Control** represents an intercranial probe inserted to a patient. Each assessment of the probe is recorded in **ICP Reading**.

The abstract class **Spinal Prec** represents any spinal precaution applied to a patient. The date/time of application and removal is recorded. The **Cervical Prec**, **Thoracic Prec** and **Lumbar Prec** represent, respectively, a cervical, thoracic and lumbar precaution applied to the patient. For the cervical precautions the information stored includes: if a hard collar or sandbags were used, if the patient's head of the bed was up 30 degrees and if the patient was prevented to lie on one side. For the thoracic and lumbar precautions no further information has to be specified.

The design of the spinal precautions hierarchy was influenced by the use of C++ in the implementation. The first approach in the design is shown in Figure 5.13. For implementing the relationship between **Hospital Gral Unit Visit** and **Spinal Prec** a set of pointers to objects of type **Spinal Prec** had to be used. The problem is that when one of these objects is removed from the set, the object's class is not known, since C++ does not know its type at runtime. This means that it is not known if we are dealing with a cervical, thoracic or lumbar precaution. Although this problem could be solved adding an extra attribute indicating the name of the class to which the object belongs to, and then casting the pointer retrieved from the set to the correct class, the space, performance and programming complexity of such solution is not justifiable. Instead, we decided to decompose the relationship in three: one for cervical, one for thoracic and one for lumbar precautions.

5.2.11 Respiratory Assessment

The *Respiratory Assessment* category contains the classes used to keep information regarding the chest exams, respiration support devices applied, ventilator control and

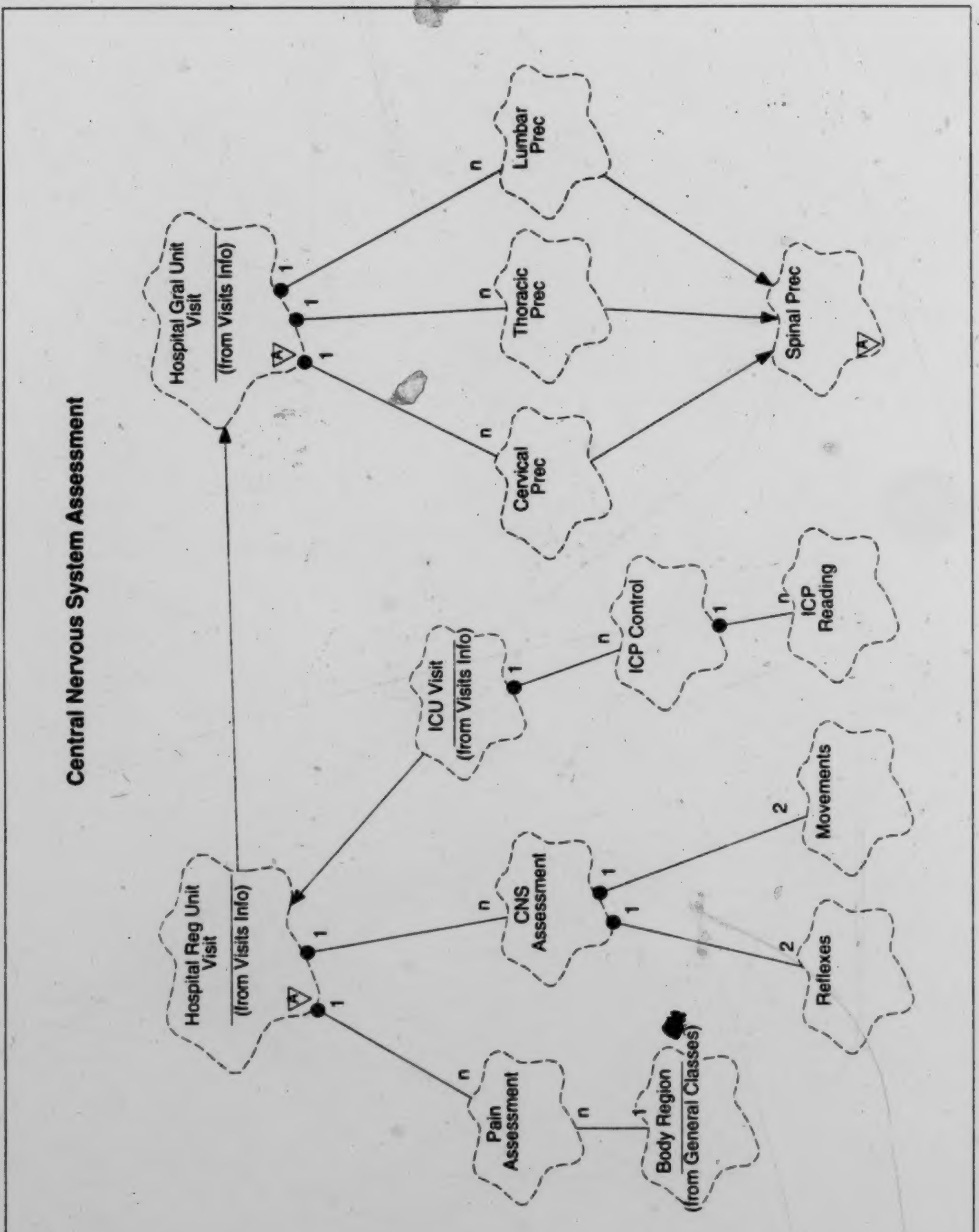
Central Nervous System Assessment
Hospital Gral Unit Visit
(from Visits Info)
Lumbar Prec
Thoracic Prec
Cervical Prec
Spinal Prec
Hospital Reg Unit Visit
(from Visits Info)
ICU Visit
(from Visits Info)
ICP Control
ICP Reading
CNS Assessment
Movements
Reflexes
Pain Assessment
Body Region
(from General Classes)
1
n
2

Figure 5.12: Central Nervous System Assessment

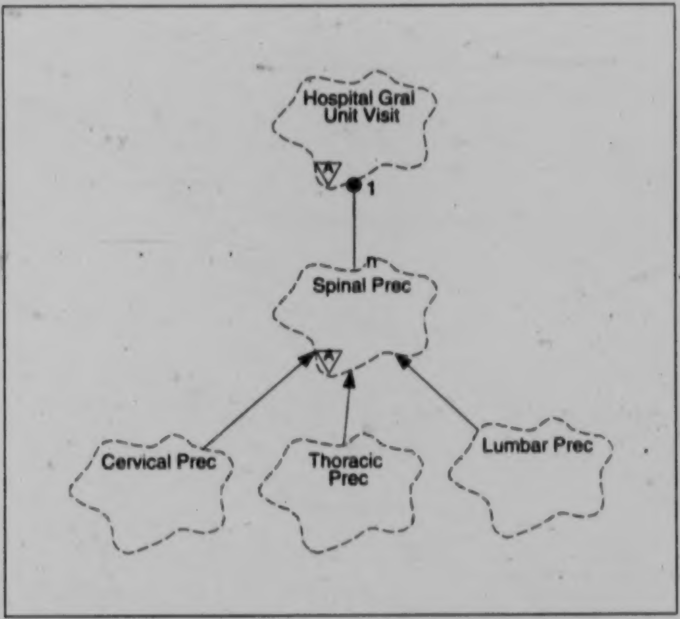

Figure 5.13: First approach in the design of spinal precautions

airway procedures done to the patients during their visits (see Figure 5.14).

For every **Chest Exam** performed on a patient in any regular hospital unit the following information has to be specified: the date/time of assessment, whether the airway is clear, obstructed or intubated, how the chest expansion is and the position of the trachea. A **Lung Exam** is also conducted each time the chest is examined, and the results of the auscultation and percussion for each of the six lobes of the lungs are recorded.

An **Airway Proc Done** represents an airway procedure that was done to a patient. For each procedure done, the type of **Airway Procedure** and the date/time when it was done is recorded. Each instance of **Respiration Support** represents a respiration support device applied to a patient. For each device the date/time of application/removal and the type of **Resp Support Device** applied have to be recorded. If a patient is under ventilator assistance, the values of the ventilator settings have to be recorded. Each instance of **Ventilator Control** represents an assessment of the settings of the ventilator at a particular point in time.

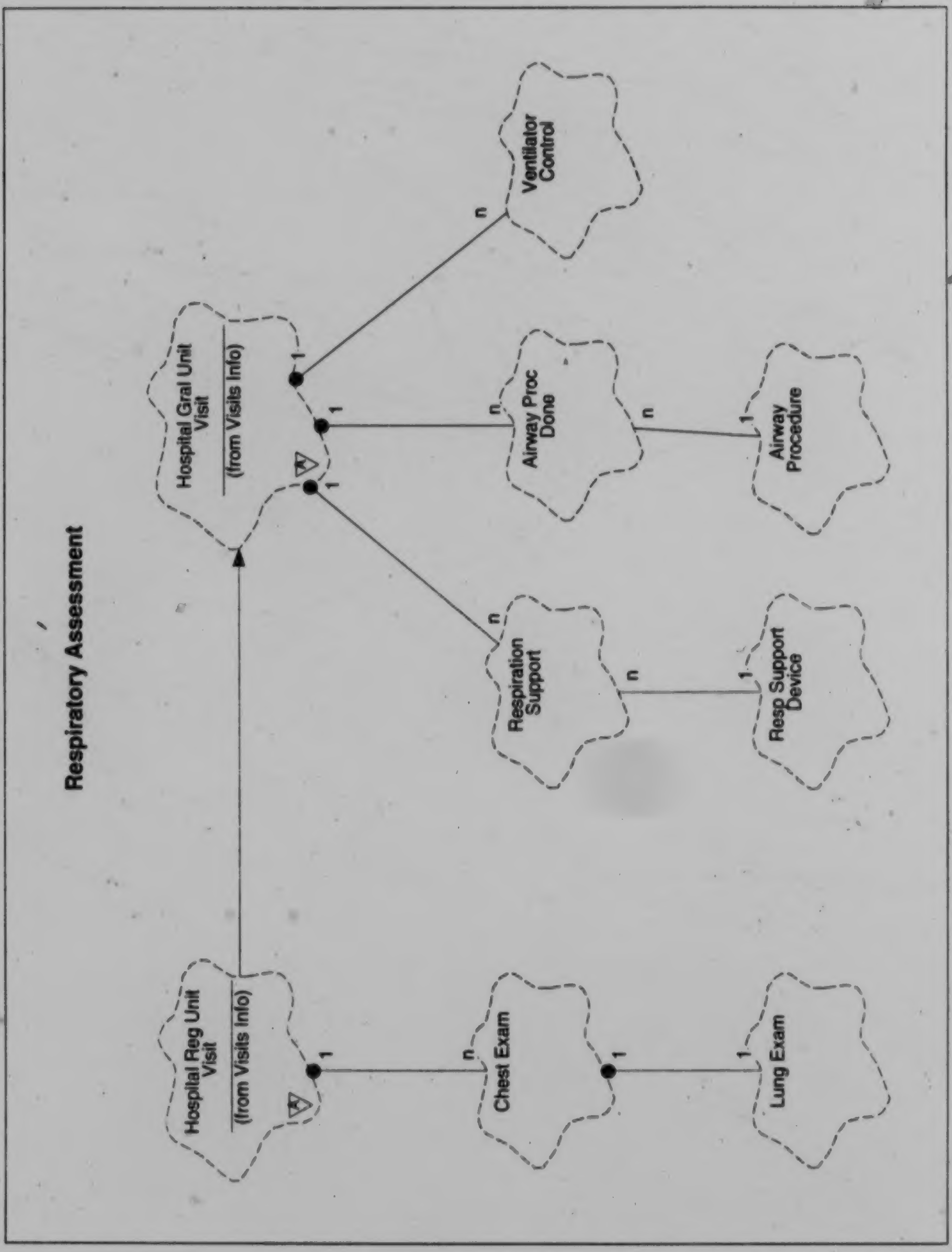
Respiratory Assessment
Hospital Reg Unit Visit
(from Visits Info)
Chest Exam
Lung Exam
Hospital Gral Unit Visit
(from Visits Info)
Respiration Support
Resp Support Device
Airway Proc Done
Airway Procedure
Ventilator Control
1
n

Figure 5.14: Respiratory Assessment

5.2.12 Vital Signs Assessment

The *Vital Signs Assessment* category contains the classes used to keep information regarding the vital signs of the patients during their visits (see Figure 5.15).

The vital signs of the patients are assessed in every unit in the hospital and the EMS. Although each unit assesses a different set of vital signs, many commonalities found leaded to the creation of superclasses.

Each instance of the abstract class **Basic Vital Signs** represents a vital signs assessment for a patient. The date/time of assessment, the body temperature, a **Pulse**, **Blood Pressure** and **Respiration** assessment are recorded. A **Pulse** assessment includes the pulse reading (i.e. heart rate) and the position, volume and rhythm of that pulse. A **Blood Pressure** assessment includes the systolic and diastolic blood pressure, the side of the body and position in which the blood pressure was assessed. A **Respiration** assessment includes the amount of breaths per second and the depth, quality and rhythm of the respiration.

During a patient's visit to the OR, the assessment of these basic vital signs is sufficient. Each instance of the **Or Vital Signs** represents a vital signs assessment for a patient during his/her visit to the OR. Although the class **Or Vital Signs** does not add any more attributes or behavior to the **Basic Vital Signs**, this class has to be created in order to associate it with an OR Visit. If the **Basic Vital Signs** were associated with OR Visit, every subclass would have inherited this relationship too, and that is not correct.

Each instance of the abstract class **Extended Vital Signs** represents a more comprehensive vital signs assessment for a patient. It inherits from **Basic Vital Signs** and adds a **Skin**, **Pupil** and **Gcs** (Glasgow Comma Scale) assessment. It also provides the calculation of the Revised Trauma Score. A **Skin** assessment includes the color, moisture, turgor, general and extremities temperature of the skin. A **Pupil** assessment includes the side of the body being assessed (i.e. left or right) and the size and response of the pupil. A **Gcs** assessment includes the Glasgow Comma Scale eye opening, verbal and motor response of the patient and provides the Glasgow Comma Scale score.

During a patient's visit to the ER, the assessment of these extended vital signs is

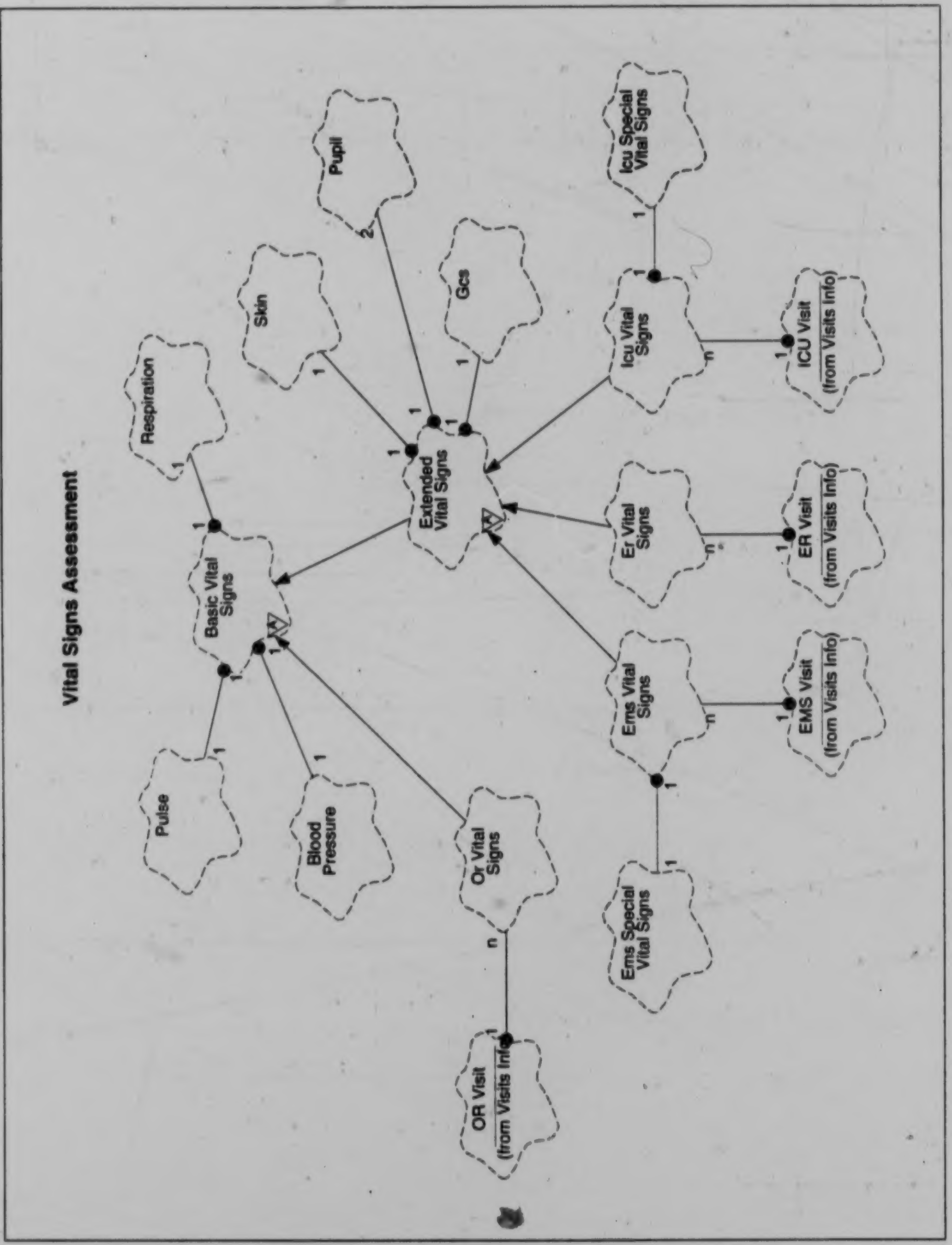
Vital Signs Assessment
Respiration
Skin
Pupil
Gcs
Icu Special Vital Signs
Pulse
Basic Vital Signs
Extended Vital Signs
Icu Vital Signs
ICU Visit (from Visits Info)
Blood Pressure
Er Vital Signs
ER Visit (from Visits Info)
Or Vital Signs
Ems Vital Signs
EMS Visit (from Visits Info)
OR Visit (from Visits Info)
Ems Special Vital Signs

Figure 5.15: Vital Signs Assessment

enough. **Ems Vital Signs**, **Er Vital Signs** and **Icu Vital Signs** each represents a vital signs assessment for a patient during his/her visit to the EMS, ER and ICU respectively.

For the ICU and EMS visits, specific vital signs for each unit are also assessed. Each instance of the **Ems Special Vital Signs** includes the glucose level and oxygen saturation of the patient and some values needed to calculate the Pre Hospital Index. Each instance of the **Icu Special Vital Signs** includes the cardiac index, the central venous pressure, the pulmonary artery pressure, the pulmonary and systemic vascular resistance index and the wedge. If a cooling or warming blanket is used it is also indicated.

5.2.13 OR Anaesthesia and Procedures

The *OR Anaesthesia and Procedures* category contains the classes used to keep information regarding the assessment of the patients prior to the operations, the operation and anaesthesia setup information, the monitor readings during the operations and the procedures done to the patients during their visits to the OR (see Figure 5.16).

Before any visit to the OR, a **Pre Assessment** of the patient needs to be done. The information recorded includes: the date/time of assessment, the class of risk of the operation, the amount of blood units available, the date/time of the last meal and the dental risk of the patient. If the operation is an emergency, the fact is also recorded in the pre assessment. A **Dentition Assessment**, a **Pre Airway Exam** and an **Anaesthetic History** assessment also have to be done prior to the operation. Each instance of the **Dentition Assessment** represents a patient's tooth with a certain problem (e.g. missing, capped, loose, etc). The most important information recorded in the **Pre Airway Exam** includes the anticipation of difficult intubation and the neck mobility. The **Anaesthetic History** records any previous anaesthetic problems of the patient or his/her family.

The operation and anaesthesia **Setup Info** includes: the **Gas Type**, anaesthesia **Technique** and **Monitors** and equipment used during the operation, the **Position** of the patient and how he/she was connected to the equipment (i.e. the **Circuit**). Whether the patient's eyes were tapped, lubbed and/or padded is also specified with the setup information.

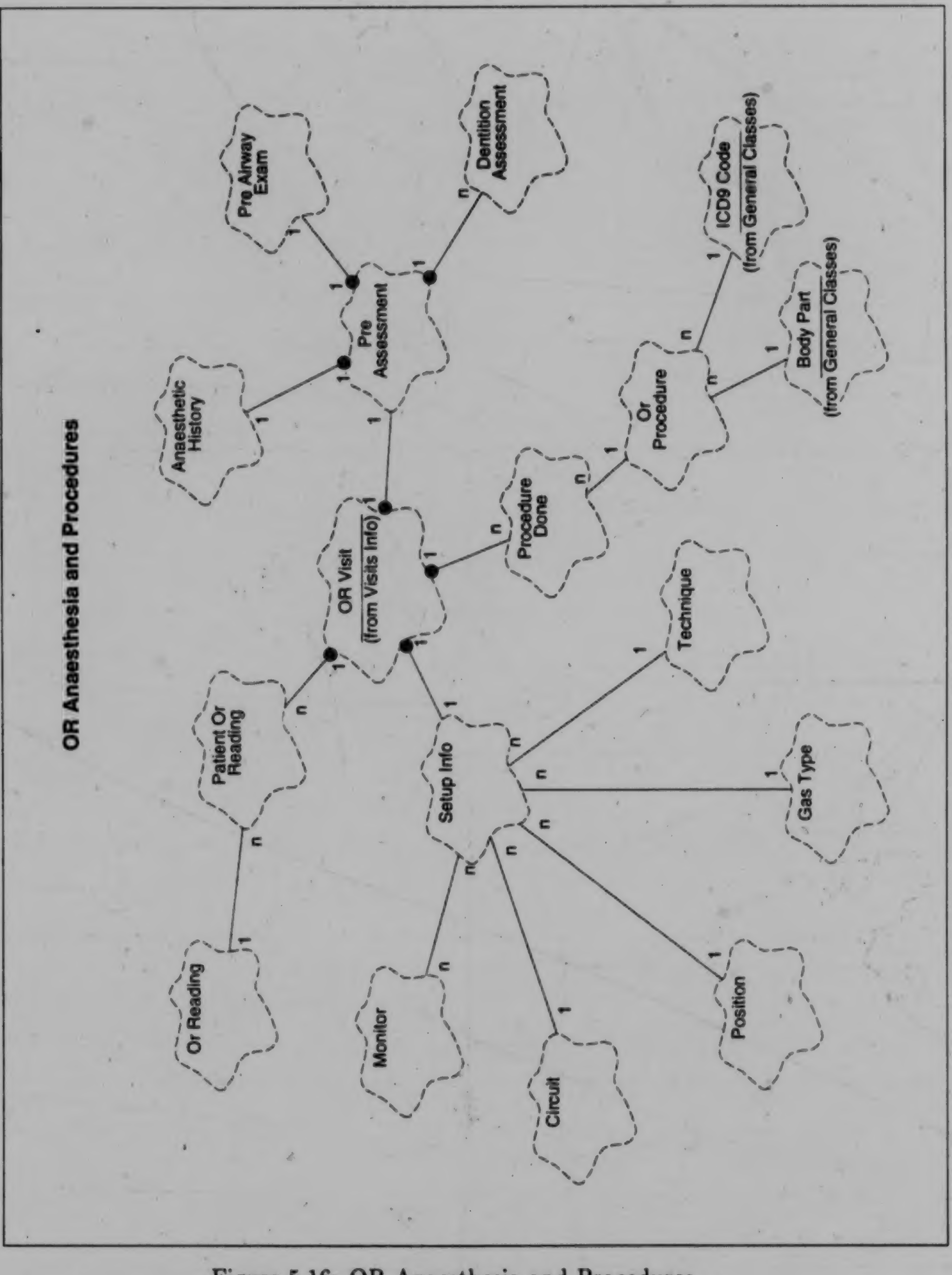
OR Anaesthesia and Procedures
Anaesthetic History
Pre Assessment
Pre Airway Exam
Dentition Assessment
OR Visit
(from Visits Info)
Procedure Done
Or Procedure
ICD9 Code
(from General Classes)
Body Part
(from General Classes)
Patient Or Reading
Or Reading
Setup Info
Technique
Gas Type
Position
Circuit
Monitor

Figure 5.16: OR Anaesthesia and Procedures

An **Or Procedure** is fully defined by a **Body Part** and **ICD9 Code**. Every **Procedure Done** to a patient during an OR Visit and all the monitor readings done during the operation have to be recorded. Each instance of **Patient Or Reading** represents one of these readings. The date/time of assessment, the type of **Or Reading** being assessed and its value are recorded.

5.2.14 EMS Specific Information

The *EMS Specific Information* category contains the classes used to keep information regarding the ambulance runs and the general assessments and treatments done to the patients during their visits to the EMS (see Figure 5.17).

Each **EMS Visit** has one ambulance run associated with it. Each instance of **Run Info** represents an ambulance run. The information recorded includes: the reason for the call, the date/time of the 911 call, the date/time of ambulance dispatch, the date/time the ambulance arrive/left the scene and the date/time of the ambulance arrival at the destination. The response level (emergency medical service, basic or advanced life support), the type of response and transport, the total kilometers of the run and the name of the policeman/woman that attend the call, if any, are also recorded. For every run, which **Vehicle** was assigned to the run, who the **Dispatcher** of the call was, the **Crew Members** in the ambulance, and the destination **Facility** are also recorded. The facilities are grouped by **Facility Type**.

If a patient is a minor or unable to give consent, a **Next Of Kin** has to authorize the transport or treatment of that patient. Every **Treatment Done** to the patient during the EMS Visit has to be recorded. The type of **Treatment** and the date/time it was done are stored.

Every possible **Diagnosis** is identified by an **ICD9 Code**. The system also records if the diagnosis is considered or not an injury, if only a specific **Body Region** applies to that diagnosis and the **Macro Diagnosis** to which it belongs. The possible **Diagnosis Modifiers** that apply for a particular diagnosis are also stored. During the **EMS Visit** the patient's condition is assessed. Each instance of the class **General Assessment** includes: the date/time of the assessment, the **Diagnosis**, the **Body Region** affected and a **Diagnosis Modifier** if needed.

We have in this diagram a problem similar to that described in Section 5.2.7. In

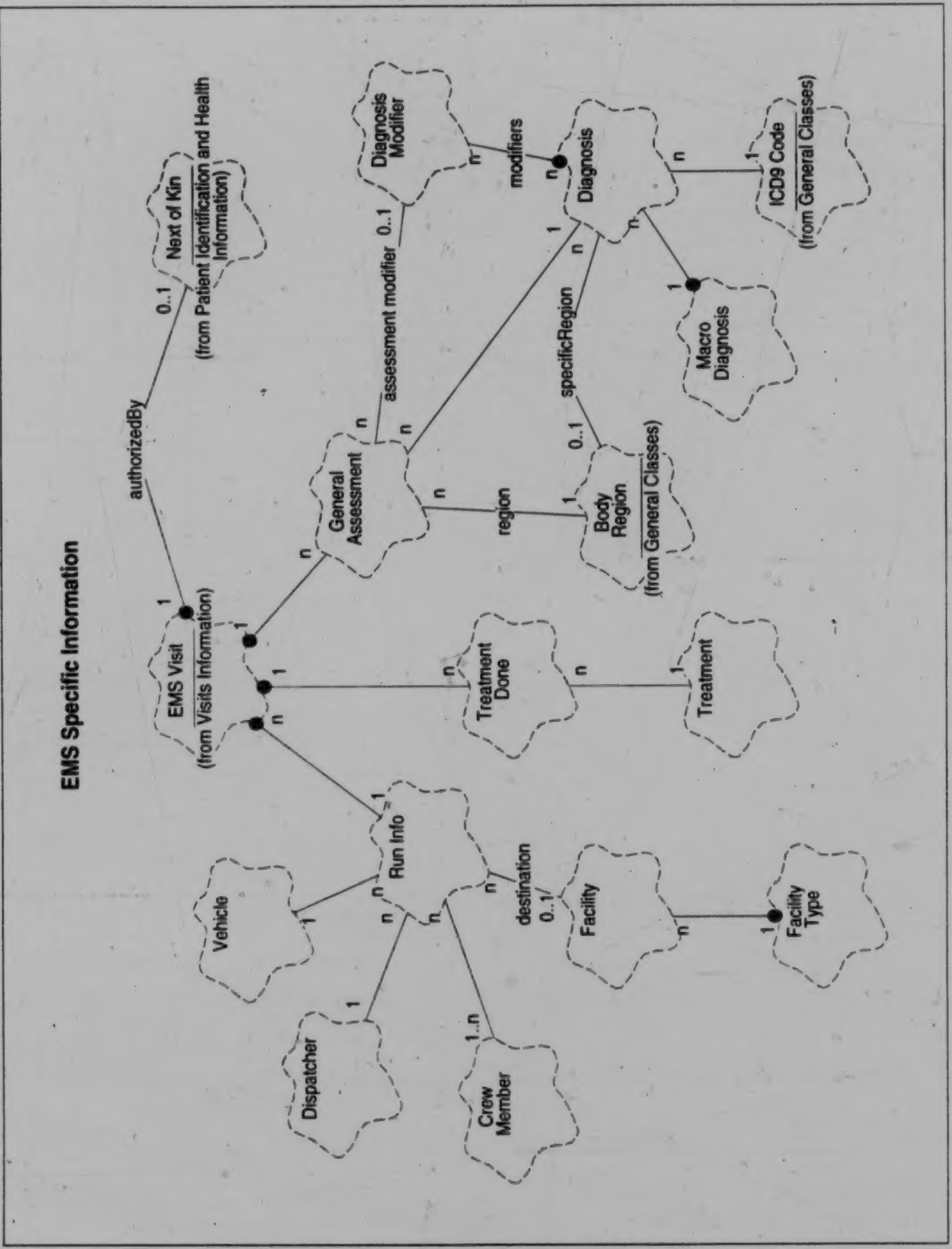
EMS Specific Information
EMS Visit
(from Visits Information)
authorizedBy
1
0..1
Next of Kin
(from Patient Identification and Health Information)
General Assessment
n
1
assessment modifier
0..1
Diagnosis Modifier
modifiers
n
Diagnosis
n
1
ICD9 Code
(from General Classes)
Macro Diagnosis
specificRegion
Body Region
(from General Classes)
region
Treatment Done
Treatment
Run Info
Vehicle
Dispatcher
Crew Member
1..n
destination
0..1
Facility
Facility Type

Figure 5.17: EMS Specific Information

this case a **General Assessment** can be associated with a **Diagnosis Modifier** if and only if that modifier belongs to the set of modifiers valid for the **Diagnosis** associated with the **General Assessment**. Introducing the same notation as before, we would need to trace a dotted arrow from the *assessment modifier* to the *modifiers* relationship. We do not present this notation in the diagram because the diagrams were generated using Rational Rose which does not support this notation or free hand drawing.

A restriction also exists for the *region* relationship. If **General Assessment** is associated with a **Diagnosis** that could only be applied to a specific **Body Region**, **General Assessment** could only be associated with that **Body Region** and no other. In any other case no restriction applies. This restriction can not be represented in the Booch notation. However, introducing a new notation for a problem that is so specific to this problem domain does not make sense. Perhaps a good idea might be to generalize the notation we introduced before so that it just represents a *restriction* between relationships without specifying which that restriction is. This will tell the person who is reading the diagram that there is a restriction and that the details are in the *Class Specifications*. Instead of a dotted arrow, we can use a dotted line with no arrow head between the two relationships with and "R" adornment meaning *Restriction*.

5.3 Summary

In this chapter a precise object-oriented model for the DIDP system is described. For each functional group identified in the requirements analysis, one class category with its correspondent class diagram is defined. The motivations behind each design decision and the influence of the implementation tools (C++/ObjectStore) in the design are discussed. A side issue in this chapter is the introduction of new elements to the Booch notation to deal with particular modeling problems. In the following chapter the issues regarding the DIDP database implementation are discussed.

Chapter 6

Implementation Issues

The design presented in Chapter 5 is sufficiently general to be implemented on most object-oriented DBMSs. In this chapter we describe the actual implementation of the DIDP database. The implementation language is C++ [Str91, Lip91]. The specific implementation of the language used is the Solaris 2.5, SPARCompiler C++ Version 4.0.1. The DBMS used is ObjectStore Version 4.0.

6.1 Overview

The database server functionality is provided though a group of libraries. There is one library per class category created in the design. Each of these libraries implements the classes of the class diagram associated with the correspondent class category.

The database server is organized as several libraries so that the application programs would only have to link-edit [1] the libraries they need. The *general* library has to be link-edited by every application program as it provides the implementation of basic and extended types, and classes that are used by most of the other libraries. Unfortunately, the idea of only link-editing some of the libraries does not work because of the way ObjectStore manages relationships. ObjectStore always needs to have the implementation of both classes that are related in order to link-edit a program. Since the classes in the *general* library have relationships with classes in almost every other library, and since the *general* library is always link-edited, this means that we also

---

[1] Link-edit is the process by which several files of machine code are combined to form a single program. These files may be the result of several different compilations, and one or more may be library files of routines provided by the system and available to any program that needs them [ASU88].

```
#include <didp.hh>
void didpInit();

int main(int, char **argv)
{
        didpInit();
        os_database *db1=os_database::open(getenv("DB"));

        OS_BEGIN_TXN(tx1, 0, os_transaction::update)

        ...

        OS_END_TXN(tx1)

        db1→close();
        return 0;
}
```

Figure 6.1: Sample application program

need to link-edit every other library related to it. Thus, all the libraries have to be link-edited to every application program that uses the DIDP database server.

Figure 6.1 shows a sample application program that uses the services of the DIDP database server. Three important things have to be noticed. First, every application program that uses the DIDP libraries has to include the **didp.hh** header. Second, the first statement in the program has to be the call to the function *didpInit()*, which performs the general initialization necessary to use the libraries and creates the database if it does not exist. Third, the path to the location of the database to be used by the application program has to be specified in an environment variable called **DB**. The *didpInit()* function checks if an ObjectStore database exists in the **DB** location, and if it does not, it creates a new one. The application programmer is responsible for opening/closing the database and for defining the transaction boundaries of the application. For details on how transactions work in ObjectStore see [Obj95].

```
...
OS_BEGIN_TXN(create_dictionary,0,os_transaction::update)

if (!(db1→find_root("Didp_root"))) {
        db1→create_root("Didp_root")
        →set_value(&os_Dictionary<String,void*>::create(db1,200,
        os_Dictionary<String,void*>::signal_dup_keys |
        os_Dictionary<String,void*>::pick_from_empty_returns_null));
}

OS_END_TXN(create_dictionary)
...
```

Figure 6.2: didpInit(): how the database root is created

6.2 Database Roots and Extents

With ObjectStore, any C++ object can be made persistent and handled the same way as transient objects. Once persistent, an object can be accessed either by navigation from other persistent objects or by giving it a persistent name. These names are called *database roots*, or *entry points*.

The set of all objects that belong to a class is called the *extent* of the class. ObjectStore does not automatically maintain the extents of classes; they have to be maintained manually. Extents are essential for queries which search over a particular class. The DIDP database server maintains the needed extents automatically as persistent parameterized sets.

Once an extent is created there must be a mechanism to locate it. That is why each extent is usually associated with a database root. In the DIDP database, there are 138 classes and extents of 50 of them have to be maintained. This poses a problem since ObjectStore recommends not to have more than 10 database roots because of performance reasons. The problem is solved in the DIDP database using a *dictionary* as the *only* database root. The key element of the dictionary is a **String** that represents the name of a class, and the second element is a pointer-to-void that points to the persistent parameterized set that represents the extent of that class.

The database root and the dictionary are created when the database is first created in the *didpInit()* function (see Figure 6.2). No duplicate values are allowed in the key element of the dictionary so that every class has one and only one extent.

With this approach, the extent of any class in the schema can be found simply by retrieving the database root, called *Didp_root*, and looking into the dictionary for the name of the class. Each class is responsible for creating its own extent and adding it to the dictionary. Also, every class that maintains its extent should provide a static method that returns this extent. A sample method for retrieving the extent of a class called *ClassName* is shown in Figure 6.3.

6.3 Basic and Extended Types

The design assumes the existence of certain basic data types. Some of these types are provided by C++, others by ObjectStore and others had to be implemented. The types **Int**, **Unsigned Int** and **Real** are implemented using the C++ **int**, **unsigned int** and **double** types respectively. The type **Bool** is implemented using the ObjectStore **os_boolean** type.

Date, **Time** and **String** had to be implemented. **Date** represents any date between 1.1.1753 and 31.12.9999. **Time** represents any point in time starting at 1.1.1902 at 00:00:00 hours. It considers day time savings and the different time zones. **String** represents any character string. Each of these classes provides a group of constructors, print methods, a method indicating if the object is null, a set of arithmetic operators and specific methods related to the class. The default constructors for **Date** and **Time** create the current date and time respectively. The default constructor for **String** creates an empty character string. **Date** provides a validation method that returns true if the parameters passed are valid for creating a date, or false if not. If invalid parameters are passed to the constructors a null date is created.

The *extended* types **Age**, **Pname** and **Address** are implemented similar to the basic data types. They provide a set of constructors, print methods, a method indicating if the object is null, a set of arithmetic operators, one or more validation methods for the constructor parameters and specific methods related to the class. If

```
os_Set<ClassName*>* ClassName::getExtent(os_database *dbl)
{
        //Check if the database exists and is open
        assert(dbl→is_open());

        // Retrieve the database root
        os_Dictionary<String,void*>* DidpExtents =
                (os_Dictionary<String,void*>*)
                dbl→find_root("Didp_root")→get_value();

        // Retrieve the extent of the class
        os_Set<ClassName*> *TheExtent = (os_Set<ClassName*>*)
                DidpExtents→pick("ClassName");

        // If the extent does not exist, create it
        if (!TheExtent) {
                os_Set<ClassName*>& extent =
                        os_Set<ClassName*>::create(dbl,
                        os_collection::pick_from_empty_returns_null |
                        os_collection::maintain_cursors);
                DidpExtents→insert("ClassName", &extent);
                TheExtent = &extent;
        }

        // Return the extent of the class
        return TheExtent;
}
```

Figure 6.3: A method for retrieving the extent of a class

invalid parameters are passed to the constructors a null object is created.

6.4 The Class Interfaces

Every class in the DIDP database, except the basic and extended types described in the previous section, has a similar interface. Obviously, each class has specific needs and thus specific methods, but there are many commonalities among the class interfaces. In this section we describe this common interface and the motivation behind each implementation decision.

6.4.1 Object Creation and Validation

When a constructor of a class is invoked in C++, the memory needed by the object is already allocated, that is, the object is already created. ObjectStore overrides the *new* method provided by C++ so that the objects that belong to classes marked as persistent are created in persistent memory (i.e. the database). This means that by the time a constructor is called, there is an object created in the database. In order to maintain consistency in the DIDP database, the parameters passed to initialize the attributes of the object need to be validated to ensure that the created object has values that do not put the database in an inconsistent state. One solution might be to return a success/failure code so that the programmer deletes the created object if it is inconsistent. This would be a simple but not an efficient solution. Anyway, C++ does not allow constructors to return any value. The application programmer would have to inspect the object to see whether or not it consistent. This means that the application programmer should know the validation procedures for each class, and thus, the advantages of encapsulation of object-oriented programming would be useless. Another solution is to provide a static validation method. The programmer would have to invoke the validation method and according to the result decide whether to invoke the *new* method or to display an error message. In this case the programmer does not need to be aware of the validation procedure but he/she is still in charge of maintaining the consistency of the database. The programmer can create an inconsistent object and put the database in an inconsistent state on purpose or by mistake. It is preferable to remove this responsibility from the programmer. For

```
Color* Color::insert(os_database *db1, const String& colorname)
{
        // Check if the database exists and is open
        assert(db1→is_open());

        // Validated the parameters
        if (Color::valins(db1,colorname) ≠ OK)
                return NULL;

        // Creates and initializes the object
        Color *element = new(db1, Color::get_os_typespec())
                Color(colorname);
        assert(element ≠ 0);

        // Inserts the new object in the extent
        os_Set<Color*> *extent = Color::getExtent(db1);
        extent→insert(element);

        // Returns a pointer to the new object
        return element;
}
```

Figure 6.4: An insert() method for the class Color

this reason, in order to create objects in the DIDP database, every class provides a static method called *insert()*. This is the only way to create objects in a class, as the class constructors are hidden as protected methods. The *insert()* method has as parameters a reference to the database in which the object is going to be created and the values necessary to initialize the attributes of the object. If the database exists and is open, and the values are valid, the object is created (i.e. the *new* method and constructor are invoked). If the class maintains its extent, the object is also added to the class extent. The *insert()* method returns a pointer to the created object. If no object was created because the validation failed, it returns a null pointer.

The problem with this approach is that if there is an error and the insert returns a null pointer, the programmer would not know what the problem was. If the *insert()*

```
Error* Color::valins(os_database *db1, const String& colorname)
{
        // Check if the database exist and is open
        assert(db1→is_open());

        // Check that the parameter is not null
        Error *err = NULL;
        if (colorname == "") {
                err = new Error(NULLVAL, INS, "ColorName");
                assert(err ≠ 0);
                return err;
        }

        // Check that the color is not already created
        if (Color::exists(db1,colorname)) {
                err = new Error(DUPLICATED, INS, "ColorName");
                assert(err ≠ 0);
                return err;
        }

        // Returns an Ok code
        return NULL;
}
```

Figure 6.5: A valins() method for the class Color

method returns an error code, the programmer would not have a pointer to the created object and would not been able to manipulate that object without having to query the database to find the object. The solution is to provide a static method called *valins()* for every class, whose return type is a pointer to **Error**. This method has exactly the same parameters as the insert method and does the validation of the parameters. If an error is found, it creates an instance of the class **Error** and returns a pointer to it, if not, it returns a null pointer.

Every possible error is codified. Every instance of **Error** stores the *ErrorType*, the *Operation* that caused the error (i.e. insert, modify, delete) and the *Attribute* for which the invalid value was intended to. It also provides a *printLine()* method that the application programmer could use to show the error to the final user. Figures 6.4 and 6.5 show the *insert()* and *valins()* methods for the class **Color**.

With these two static methods we ensure the consistency of the DIDP database and free the application programmer to concentrate on the semantics of his/her programs instead of taking care of the consistency of the database.

6.4.2 Object Deletion

Problems similar to those found in object creation were found for object deletion. When an object invokes its destructor, that object is deleted. In order to maintain consistency in the DIDP database, we need to be sure that no object is pointing to the object we want to delete. The problem is not dangling pointers, ObjectStore takes care of that assigning null to the pointers that are pointing to the deleted object. The problem is the logical consistency of the database. For example, every injury is uniquely identified by a **Body Region** and an **ICD9 Code**. If after creating an instance in **Injury** we delete the **ICD9 Code** that is associated with it, the injury definition would be incomplete and inconsistent, as it must be associated with an ICD9 code, and thus the whole database would be in an inconsistent state. Because this validation must be made *before* deleting the object, it can not be made in the class destructor, as once the destructor execution starts there is no way to cancel the deletion operation. For this reason, in order to delete objects from the DIDP database, every class provides a static method called *erase()*. This is the only way to delete objects from the database as the destructor of the class is hidden as a protected method. If an error is found during the validations, an **Error** object is created and a pointer to it is returned. If no error is found, the object is deleted (i.e. its destructor is invoked) and a null pointer indicating that the operation was successful is returned.

6.4.3 Attribute Retrieval and Modification

Every attribute of every class is **private**. This means that no method/program can access the attributes of any class except through its interface. Every attribute that a class wants other classes/programs to see has a *getAttributeName()* method that returns the attribute value. Also, every attribute that the class allows to be modified has a *modAttributeName()* associated. This modification method validates the new value before modifying the attribute in the database. If an error is found during the validation, an **Error** object is created and a pointer to it is returned. If no error is found, the new value is assigned to the attribute and a null pointer indicating that the operation was successful is returned.

6.4.4 Classes with Extent

We can divide the classes in the DIDP database into two categories: the *Passive* classes and the *Active* classes. *Passive* classes are those classes whose instances represent things from the real world that have no relevance in the system unless they are referenced by an object that belongs to an *Active* class. The extents of the *Passive* classes must exist before the system can be used. For example, for every native patient, the band to which he/she belongs to has to be specified. Each instance of the class **Band** represents a different native band. When an instance of **Patient** is created to represent a native patient, the object that represents the **Band** to which the native belongs to has to exist in order to reference it from the **Patient** object. The native bands by themselves have no relevance to the system, but determining which band each patient belongs to, is relevant. **Band** is a *Passive* class and **Patient** is an *Active* class.

The extent of every passive class and the extents of the classes **Patient** and **RunInfo** are maintained by the DIDP database server to facilitate queries. Every other class can be accessed navigating through other objects.

All the classes whose extents are maintained have four standard static methods: *getExtent()*, *exists()*, *get()*, and *printClass()*. *getExtent()* returns a pointer to the set that contains pointers to all the objects that belong to the class extent. *exists()* returns true if an object that contains the values passed as parameters exists in the class extent and false otherwise. *get()* returns a pointer to the object that contains the values passed as parameters if any exist in the class extent. If none is found, a null pointer is returned. *printClass()* prints all the objects in the extent to the output stream passed as a parameter.

6.4.5 Relationships

Relationships were implemented using the mechanisms ObjectStore provides. ObjectStore supports *One-to-One*, *One-to-Many* and *Many-to-Many* relationships. It represents relationships using pointers and collections. *One-to-One* relationships are represented by a pointer on each side. *One-to-Many* and *Many-to-One* relationships are represented by a collection from the **one** side and a pointer from the **many** side. *Many-to-Many* relationships are represented by collections on both sides. ObjectStore relationships ensure referential integrity [2] for participating objects. When code modifies one side of the relationship the other side is automatically updated by ObjectStore therefore guaranteeing referential integrity. Although this integrity can be maintained by the programmer via explicit code in the applications, ObjectStore takes care of it saving time and preventing programming errors.

The DIDP database server keeps the relationship members of the classes private. Every class provides the correspondent *get()* methods for those relationships members that want other classes/programs to see.

One-to-Many and *Many-to-One* relationships are always created by passing a pointer to an object of the class on the **one** side to the *insert()* method of the object in the **many** side. For example, in order to create an instance of **PatientAllergy**, a pointer to **Patient** is required in its *insert()* method. This method will invoke the constructor of the class and will create the link between the two objects. ObjectStore takes care of the inverse relationship. This means that a pointer to the **PatientAllergy** object will be automatically inserted in the collection maintained by the **Patient** object to represent the relationship. If the **PatientAllergy** object is later deleted, ObjectStore will remove the pointer from the collection in **Patient** so that no dangling references are left.

One-to-One relationships work in a similar manner. One of the classes in the relationship is chosen to require a pointer to an object in the other class in its *insert()* method. The rest of the mechanism is the same as the one described above.

When a *Many-to-Many* relationship exists between two classes, only one of the classes in the relationship is responsible to create the links between the objects of the two classes. For this purpose, this class provides an *addClassName()* and a *removeClassName()* methods that create and remove the relationship between the two. For example, **Patient Valuables** has a *Many-to-Many* relationship with **Valuable**. **Patient Valuables** provides two methods called *addValuable()* and *removeValuable()* that create and delete the links between these two classes.

---

[2] The term *integrity* refers to the accuracy or validity of data. To ensure *referential integrity* means to ensure that no object has a reference (i.e. pointer) to a non-existent object.

6.4.6 Printing Methods

Every class provides a *printLine()* and a static *printSet()* method. *printLine()* prints the object to the output stream passed as a parameter. The static method *printSet()* prints all the objects in the set passed as a parameter to the output stream indicated. The set passed as a parameter is a set of pointers to objects that belongs to the class to which the *printSet()* method belongs to. *printSet()* is used to implement the *printClass()* method.

6.5 Other Issues

In this section we describe some particular problems found during implementation due to the implementation tools used (i.e. C++/ObjectStore) and how they were solved.

6.5.1 The Visits Hierarchy Problem

The visits inheritance hierarchy is created to group all the attributes and behavior that the **EMS Visit**, **ER Visit**, **OR Visit** and **ICU Visit** classes have in common. As depicted in Figure 6.6, **Patient Visit** has a *One-to-Many* relationship with the **Unit Visit** class. This means that every patient can visit many units during his/her visit to the hospital/EMS. As described before, a *One-to-Many* relationship is implemented using a pointer on the **many** side and a set of pointers in the **one** side. Thus, every object that belongs to **ER Visit**, **OR Visit**, **EMS Visit** and **ICU Visit** [3] has a pointer to one object of **Patient Visit**, and every object of **Patient Visit** has a set of pointers of type **Unit Visit**.

---

[3] These are the only non-abstract classes in the hierarchy, and thus, the only ones that can be instantiated.

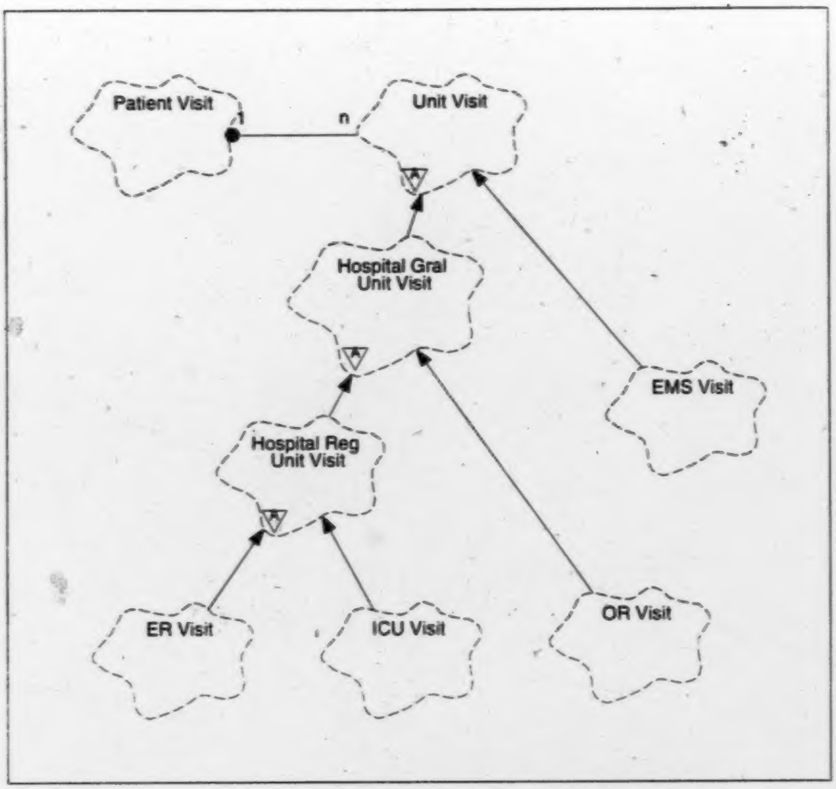
Patient Visit
1
n
Unit Visit
Hospital Gral Unit Visit
EMS Visit
Hospital Reg Unit Visit
ER Visit
ICU Visit
OR Visit

Figure 6.6: The visits hierarchy

The problem is that we only know that **Patient Visit** has a set of pointers of type **Unit Visit**, but we do not know at runtime if a pointer is a pointer to an object of type **ER Visit**, **ICU Visit**, **OR Visit** or **EMS Visit**. This happens because C++ objects do not know their type at runtime.

One solution would be to create four relationships: one for **ER Visit**, one for **ICU Visit**, one for **OR Visit** and one for **EMS Visit**. Although this approach is used for the spinal precautions (see Section 5.2.10), it can not be used in this context because there are some methods in **Hospital Reg Unit Visit**, **Hospital Gral Unit Visit** and **Unit Visit** that need to know to which **Patient Visit** the object belongs to, and if the relationship is in the leaves of the hierarchy the superclasses are not aware of it. For example, in the **Hospital Reg Unit Visit** we need to validate that the date/time that the patient enters/leaves a unit is between the date/time the patient enters/leaves the hospital, and this data is stored in **Patient Visit**.

To solve this problem an extra attribute is added to **Unit Visit** called *Unit*. This attribute indicates to which class the object belongs to and is initialized with the correct value when the object is created. Although the attribute is private, the class interface provides a *getUnit()* method that returns its value. Each time a pointer is retrieved from the set in **Patient Visit**, the programmer only has to ask to which class the pointer belongs and cast the pointer to the correct type. In order to avoid this inconvenience to the application programmer, the **Patient Visit** class provides four methods called *getErVisit()*, *getIcuVisits()*, *getOrVisits()* and *getEmsVisits()* that take care of the casting and return the set of **ER Visit**, **ICU Visit**, **OR Visit** and **EMS Visit** respectively. With this approach this implementation problem is completely hidden from the application programmer.

6.5.2 Static Functions in C++

Many static methods are used in the DIDP database schema (e.g. *insert()*, *getExtent()*, etc.). The implementation of many of these methods are very similar from one class to the other.

For example, the implementation of *getExtent()* is exactly the same for every class except that there are some **os_Set** declarations in its body that need a parameter indicating the type of the objects in the set. This parameter should be the name of the class to which the method is bound at runtime. For example, for *Color::getExtent()*, the **os_Set** declarations would look like: **os_Set<Color*>**. Unfortunately there is no way to do this in C++ static functions.

The other problem posed by static functions in C++ are the calls to other static functions. Assume that there are two static methods *MethodA()* and *MethodB()* that belongs to **ClassX**, where *MethodA()* invokes *MethodB()* within its body. Further assume that **ClassX** has a subclass, **ClassY**, that redefines *MethodB()*. When *ClassY::MethodA()* is invoked, we expect this method to invoke the redefined *MethodB()*. Unfortunately, this is not the case. The *MethodB()* from **ClassX** is invoked.

Since the implementation of many of these methods are so similar, duplicating the code is not an efficient solution. Any change that has to be done to one of these methods in the future would mean changing *every* implementation in *every* class. In order to solve this problem we used *macros*. Thus, although every class has to declare the method in its header, there is a unique implementation. Among the functions implemented as macros, we have: *getExtent()*, *get()*, *exists()*, *printClass()*, *printSet()* and *insert()*. This approach saves hundreds of lines of code and simplifies the future changes to the methods.

6.5.3 Cascade Deletion

Every time a **Patient** object is deleted from the database, we also want to delete all the information related to him/her. That means that all the objects that belong to the **Patient** object have to be deleted as well. ObjectStore provides a mechanism that automatically takes care of this issue. The problem is that it requires the destructors of the classes to be public. As explained in Section 6.4.2, the DIDP database server keeps the destructors of the classes protected. For this reason, this ObjectStore facility is not used, and the cascade deletion was implemented manually. Each *erase()* method invokes the *erase()* methods of the objects it owns before invoking its own destructor, and thus achieving the required cascade deletion.

6.6 Summary

In this chapter the implementation details of the DIDP database server are discussed. An explanation on how the libraries that support the database server functionality are organized, and a sample on how the application programs that use the services of the DIDP database server should look like, are provided. The solution of the database root problem and the handling of class extents is also explained. Some details on the implementation of basic and extended types are discussed. Also, a detailed description of the class interfaces is provided. Some problems found due to the implementation tools used, and how they were solved are also mentioned at the end of the chapter.

Chapter 7

Conclusions and Future Work

This thesis describes the analysis, design and implementation of an object-oriented database server for the Dynamic Injury Data Project (DIDP). This server provides persistent storage of the data, ensures its integrity, and provides a mechanism for the applications to interact with the data.

The major contributions of this thesis can be summarized as follows:

- The analysis of the requirements necessary to develop an object-oriented database server for the the DIDP system. The constraints specific to the problem domain were identified during this process.

- The definition of the minimum data sets for injury surveillance in each of the hospital units and the emergency medical services.

- The definition of the initial system and data processing architectures for the DIDP system. Since the database server is designed and implemented as independent as possible from these architectures, it could be easily extended or modified in the future if needed.

- The design of a detailed object-oriented model for the DIDP database. The model is sufficiently general to be implemented in any object-oriented DBMS. The documentation for the model is stored in Rational Rose so that it could be easily modified if changes are needed.

- The introduction of new elements in the Booch's notation to deal with particular modeling problems found during the model definition.

- The design and implementation of the database server mechanisms to ensure the consistency and encapsulation of the data.
- The successful implementation of the database server in the form of a library to enable access of application programs and end users to the data.

This work is only the initial step towards the development of the DIDP system. In the future, there are a number of modifications and enhancements that can be introduced:

- **Multimedia data.**

  The current version of the database server only supports text and regular formatted data. Nevertheless, the database server is designed considering the requirements needed to include multimedia data. For example, the complex nature of multimedia data was considered when choosing the database model; that is one of the reasons for selecting an object-oriented approach. In the future, multimedia data such as the images ordered for the patients (e.g. X-rays, CT Scans) and video of the scenes of the incidents, could be easily added to the database.

- **Global positioning system (GPS).**

  This feature will allow to exactly locate where an injury occurred. There are many ways of collecting the GPS information. In urban settings, the system could ask for the address where the incident took place and translate this address into geographic coordinates. In rural areas where an address could not be entered, GPS units could be used to identify the location. There are two ways of implementing the use of GPS units. One is to install hand-held GPS units in the ambulances and have the EMS personnel take note of the location of the injury and enter the information into the computer. The second one is to interface the GPS units directly with the computer to entire eliminate the manual entry of the GPS information. Both have their pros and cons. The first solution is easier to implement and less expensive, but could introduce errors. The second solution is more expensive but prevents errors in entering the data.

Despite the solution adopted. GPS information will be very useful in developing injury intervention strategies.

- **Visual query interface.**

  A visual query interface should be developed in order to allow end users to perform queries in an easy and friendly manner. In particular, the support of content-based queries of images and video would open endless possibilities in the study of injuries. For example, the automatic recognition of certain patterns in an X-ray could help with the diagnosis of certain conditions.

- **Performance optimization.**

  In the current version of the database server, ObjectStore clustering and indexing is not considered, as the type of queries that would be regularly performed is not yet identified. During the pilot testing of the database, access patterns should be studied in order to add optimization mechanisms to improve performance.

- **Improvement of the audit mechanism.**

  The audit mechanism provided by the current version of the database server has several limitations. First, only the last user that makes a modification on the object is recorded instead of having detailed information of the evolution of the operations. Second, when an object is modified, the older values of the object are lost. It could be useful to know not only that a modification was done, but exactly which changes were made to the data. The last problem is that when an object is deleted, all the information including the audit information is lost. A solution might be to make only a logical deletion of the object, marking it as deleted, and offering the database administrator a process to physically delete the objects once that the audit information has been used. A more comprehensive audit mechanism has to be implemented in a future version of the database server in order to solve these problems. The complexity of such a system will be determined by the need of the DIDP system to audit its data.

- **Security features.**

  Maintaining patient confidentiality is a very important issue. In order to limit access to the database server a fingerprint mechanism will be implemented. Access will then be monitored by the system requiring personnel to have their fingerprints electronically digitalized. A fingerprint scanner will verify the user identity. Several levels of security will be provided for different users.

- **Voice data collection.**

  Although using hand-held pen-based computers greatly simplifies the data collection, there are environments where voice data collection could be useful. An example might be the data collection in the EMS where the paramedics could collect the data at the same time that they are providing treatment to the patient. The data could be "dictated" to the computer using voice commands and *key* words that the computer would have pre-stored. The use of voice data collection could greatly improve the collection of data in specific environments.

- **Direct connection of medical equipment to the DIDP system.**

  Instead of having the nurses collecting data from devices that are connected to the patients and entering manually these data into the DIDP database, the devices connected to the patient could interface with the DIDP system sending the necessary changes in the patients' condition automatically when needed.

- **Treatment guidelines.**

  Instead of having the common practices for injury treatment loaded into the DIDP database, the database server should be able to retrieve the needed information from existing repositories. Once an injury is diagnosed, the server will be able to show the doctors the latest procedures that have to be followed for treating that particular injury. This interconnection can be accomplished either by simple triggers or by sophisticated AI-based learning techniques.

Bibliography

[ABD+89] Atkinson M., Bancihon F., DeWitt D., Dittrich K., Maier D. and Zdonik S., "The Object-Oriented Database System Manifesto", Proceedings of 1st International Conference on Deductive and Object-Oriented Databases (DOOD), Kyoto, Japan, December 1989.

[ASU88] Aho A.V., R. Sethi, J.D. Ullman, "Compilers, principles, techniques and tools", Addison-Wesley, 1988.

[BD90] Ball M.J. and Douglas J.V., "Healthcare Informatics", Healthcare Informatics Magazine, May 1990.

[BM93] Bertino E. and L. Martino, "Object-Oriented Database Systems: Concepts and Architectures", Addison-Wesley, 1993.

[Booch93] Booch G., "Object-Oriented Analysis and Design with Applications", Benjamin/Cummings, 1993.

[EN89] Elmasri R., S.B. Navathe, "Fundamentals of Database Systems", Benjamin/Cummins, 1989.

[Eval88] Centers for Disease Control, "Guidelines for Evaluating Surveillance Systems", MMWR, Vol.37, No.S-5, May 1988.

[FSH91] Francescutti L.H., L.D. Saunders and S.M. Hamilton, "Why are there so many injuries? Why aren't we stopping them?", Canadian Medical Association Journal, 144(1), pp.57-61, 1991.

[Fra97] Francescutti L.H., "Injury Control: Are you accountable?", The Canadian Journal of CME, pp.109-119, January 1997.

[GRT+94] Garrison H.G., C.W. Runyan, J.E. Tintinalli, C.W. Barber, W.C. Bordley, S.W. Hargarten, D.A. Pollock and H.B. Weiss, "Emergency Department Surveillance: An Examination of the Issues and a Proposal for a Nation Strategy", Annals of Emergency Medicine, 24(5), pp.849-855, 1994.

[GS90] Greenes R.A. and Shortliffe E.H., "Medical Informatics: An Emerging Discipline with Academic and Institutional Perspectives", Journal of the American Medical Association 263(8):1114-1120, 1990.

[Jac92] Jacobson I., "Object Oriented Software Engineering, A Use Case Driven Approach", Addison-Wesley, 1992.

[Lip91] Lippman S.B., "C++ Primer", 2nd Edition, AT&T Bell Laboratories, December 1991.

[Mac84] MacKenzie E.J., "Injury Severity Scales: Overview and Directions for Future Research", American Journal of Emergency Medicine. Vol.2, No.6, pp.537-549, 1984.

[Obj95] ObjectStore Release 4.0, Object Design Inc., C++ API User Guide, June 1995.

[Obj97] Object Design, Inc. Home Page, http://www.odi.com/AboutObjectDesign/

[PM89] Pollock D.A. and P.W. McClain, "Trauma Registries: Current Status and Future Prospects", Journal of The American Medical Association, Vol.262, No. 16, October 1989.

[RBB92] Runyan C.W., J.M. Bowling and S.I. Bangdiwala, "Emergency Department Record Keeping and the Potential for Injury Surveillance", The Journal of Trauma, Vol.32, No. 2, 1992.

[RRTB92] Ribbeck B.M., Runge J.W., Thomason M.H., Baker J.W., "Injury Surveillance: A Method for Recording E Codes for Injured Emergency Department Patients", Annals of Emergency Medicine, 21:37-40, January 1992.

[Rose95] Rational Rose Release 2.7, Rational Software Corporation, Using Rational Rose/C++ documentation set, May 1995.

[Rose97] Rational Software Corporation Home Page, http://www.rational.com/pst/products/rosecpp.html

[Sch96] Schöne M., "A Generic Type System for an Object Oriented Multimedia Database System", Master's thesis, University of Alberta, Department of Computing Science, 1996.

[Str91] Stroustrup B., "The C++ Programming Language", Addison-Wesley, 1991.

[TC94] S.M. Teusch and R.E. Churchil (editors), "Principles and practice of public health surveillance", Oxford University Press, 1994.

[WFP96] Williams J.M., P.M. Furbee and J.E. Prescott, "Development of an Emergency Department-Based Injury Surveillance System", Annals of Emergency Medicine, 27:1, January 1996.

[WFPP95] Williams J.M., P.M. Furbee, J.E. Prescott and D.J. Paulson, "The Emergency Department Log as a Simple Injury Surveillance Tool", Annals of Emergency Medicine, 25:5, May 1995.

[WG94] White I., M. Goldberg, "Using the Booch Method: A Rational Approach", Benjamin/Cummings, 1994.

Appendix A

Class Specifications

The following are the class specifications for the DIDP model. The classes are grouped by category and inside each category they are shown alphabetically. The order of the categories is:

1. Incidents and Personnel
2. Central Nervous System Assessment
3. Visits Information
4. Other Assessments
5. OR Anaesthesia and Procedures
6. Gastrointestinal Assessment
7. Diagnostic Images and Lab Exams
8. Patient Identification and Health Information
9. Medications, Antibiotics and Ivs
10. Invasive Therapy, Instrumentation and Fluids
11. Vital Signs Assessment
12. EMS Specific Information
13. Respiratory Assessment
14. General Classes

Class name:
Critical Incident

Category: Incidents and Personnel
Documentation:
Represents a critical incident that might occur to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Critical Incident Occured

Category: Incidents and Personnel
Documentation:
Represents a critical incident that occured to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Critical Incident Reason incident reason
We must validate that the reason chosen is in the set of reasons allowed for the critical incident chosen.

Critical Incident incident

Private Interface:
Has-A Relationships:
Time TimeIncident
Date/Time of the critical incident.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Critical Incident Reason

Category: Incidents and Personnel
Documentation:
Represents a reason why a critical incident might occur.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
Associations:

Critical Incident reasons

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Personnel Contacted

Category: Incidents and Personnel
Documentation:
Represents a personnel contacted during a patient's visit.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Personnel Type

Private Interface:
Has–A Relationships:

Pname Name
Name of the person contacted.

Time TimeCalled
Date/Time the person was called.

Time TimeContact
Date/Time the person did the contact. It must be >= TimeCalled.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Personnel Type

Category: Incidents and Personnel
Documentation:
Represents a type of personnel in the hospital.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

CNS Assessment

Category: Central Nervous System Assessment
Documentation:
Represents an assessment of a patient's central nervous system.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has–A Relationships:
Reflexes
Movements

Private Interface:
Has–A Relationships:

String Comments
Comments on the CNS assessment.

Time TimeAssessment
Date/Time of assessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Cervical Prec

Category: Central Nervous System Assessment
Documentation:
Represents a cervical spinal precaution applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Spinal Prec
Private Interface:
Has-A Relationships:

Bool HardCollar
Was a hard collar used for the cervical precaution of the patient?

Bool HeadUp
Was the patient's head of bed up 30 degrees?

Bool NoSideLying
Was the patient prevented to lie on one side?

Bool SandBags
Were sand bags used for the cervical precaution of the patient?

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

ICP Control

Category: Central Nervous System Assessment
Documentation:
Represents an intracranial pressure probe inserted to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has-A Relationships:
ICP Reading

Private Interface:
Has-A Relationships:

Time TimeInserted
Date/Time the probe was inserted.

Time TimeRemoved
Date/Time the probe was removed. It must be >= TimeInserted

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

ICP Reading

Category: Central Nervous System Assessment
Documentation:
Represents an intracranial pressure probe reading of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Operations:
Cpp ( )

Private Interface:
Has-A Relationships:

Enum DrainManipulation
Indicates the drain manipulation mode. The possible values are: Open intermittent, Open continously.

Unsigned Int IcpReading
Intracranial pressure reading. Values: 0–120 Hg.

Unsigned Int MapProbe
Mean arterial pressure measured through the probe. Values: 0–200 Hg

Time TimeAssessment
Date/Time of assessment. It must be between the D/T the probe was inserted and D/T the probe was removed.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Lumbar Prec

Category: Central Nervous System Assessment
Documentation:
Represents a lumbar spinal precaution applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Spinal Prec
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Movements

Category: Central Nervous System Assessment
Documentation:
Represents a movement assessment of a patient for one side of the body.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

MovementTy Arm
Movement of the arm.

MovementTy Leg
Movement of the leg.

SideTy Side
Side of the body assessed.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Pain Assessment

Category: Central Nervous System Assessment
Documentation:
Represents a pain assessment of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Body Region

Private Interface:
Has-A Relationships:

String Description
Description of the pain.

Enum Intensity
Intensity of the pain. The possible values are: Light, Moderate, Severe.

Unsigned Int Severity
Severity of the pain. Values: 1–10.

Time TimeAssessment
Date/Time of assessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Reflexes

Category: Central Nervous System Assessment
Documentation:
Represents a reflexes assessment of a patient for one side of the body.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security

Private Interface:
Has-A Relationships:
ReflexTy Ankle
Reflexes of the ankle.

ReflexTy Biceps
Reflexes of the biceps.

ReflexTy Knee
Reflexes of the knee.

ReflexTy Plantar
Reflexes of the plantar.

SideTy Side
Side of the body assessed.

ReflexTy Supinator
Reflexes of the supinator.

ReflexTy Triceps
Reflexes of the triceps.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Spinal Prec

Category: Central Nervous System Assessment
Documentation:
Represents a spinal precaution applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:
Time TimeApplied
Date/Time the precaution was applied.

Time TimeRemoved
Date/Time the precaution was removed. It must be >= TimeApplied

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Thoracic Prec

Category: Central Nervous System Assessment
Documentation:
Represents a thoracic spinal precaution applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Spinal Prec
State machine: No
Concurrency: Sequential

Persistence: Persistent

Class name:

Cause

Category: Visits Information
Documentation:
Represents a cause that might produce an incident.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
Private Interface:
Has–A Relationships:
String Ecode
E–code that corresponds to the cause. The picture is NNN.N where N is a number between 0 and 9.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

EMS Visit

Category: Visits Information
Documentation:
Represents a visit of a patient to the EMS.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Unit Visit
Public Interface:
Has–A Relationships:
Treatment Done
General Assessment
Run Info
Ems Vital Signs
Next of Kin authorizedBy
Person that authorized the transport or the treatment of the patient is s/he is a minor or is unable to give consent.

Private Interface:
Has–A Relationships:
String DiagnosticCode
Diagnostic code of the patient if the service uses them.

String InvoiceNo
Number assigned to the trip in the dispatch log or number of the invoice issued to patient.

String PCR
Patient Care Report Number. It can have up to 6 digits.

String ReasonForAmbulance
Most important problem the patient describes.

Pname ReceivingPhysician
Name of the receiving physician in the facility.

Enum Transfer
If patient was transported from one facility to another, indicates whether the

patient was an in–patient (currently admitted into the facility) or an out patient (not admitted). Values: None,In, Out.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

ER Visit

Category: Visits Information
Documentation:
Represents a visit of a patient to the ER.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Hospital Reg Unit Visit
Public Interface:
Has–A Relationships:
Er Vital Signs

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Hospital Gral Unit Visit

Category: Visits Information
Documentation:
Represents a visit of a patient to any unit in the hospital.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Unit Visit
Public Interface:
Has–A Relationships:
Image Ordered
Lab Exam Ordered
Respiration Support
Airway Proc Done
Ventilator Control
Injury
Musskel Assessment
Personnel Contacted
Critical Incident Occured
Antibiotic Given
Instrument Applied
Patient Output Fluid
Patient Intaken Fluid
Cervical Prec
Thoracic Prec
Lumbar Prec

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Hospital Reg Unit Visit

Category: Visits Information
Documentation:
Represents a visit of a patient to any regular unit in the hospital.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Hospital Gral Unit Visit
Public Interface:
Has-A Relationships:
GU Procedure Done
Chest Exam
CVS Assessment
CNS Assessment
Gi Exam
Pain Assessment

Private Interface:
Has-A Relationships:
Time TimeEnterUnit
Date/Time the patient entered the unit.

Time TimeLeftUnit
Date/Time the patient left the unit. It must be >= TimeEnterUnit.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
ICU Visit

Category: Visits Information
Documentation:
Represents a visit of a patient to the ICU.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Hospital Reg Unit Visit
Public Interface:
Has-A Relationships:
ICP Control
Stool Assessment
Ostomy Assessment
Icu Vital Signs

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Incident Info

Category: Visits Information
Documentation:
Information related to the environment where a patient was injured.

Export Control: Public
Cardinality: n
Hierarchy:

Superclasses: Security
Associations:

Cause
Safety Device

Private Interface:
Has–A Relationships:

Enum Setting
Indicates if the activity being done when the patient was injured was being done outdoors or indoors. Values: Outdoors, Indoors.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

OR Visit

Category: Visits Information
Documentation:
Represents a visit of a patient to the OR.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Hospital Gral Unit Visit
Public Interface:
Has–A Relationships:

Setup Info
Patient Or Reading
Procedure Done
Pre Assessment
Or Vital Signs

Private Interface:
Has–A Relationships:

Pname Anaesthetist
Name of the anaesthetist.

Time EndAnaesthesia
Date/Time anaesthesia ended. Must be >= TimeAnaesthesiaStart.

Time EndOperation
Date/Time operation ended. Must be >= TimeOperationStart.

Time StartAnaesthesia
Date/Time anaesthesia started.

Time StartOperation
Date/Time operation started. Must be >= TimeAnaesthesiaStart.

Pname Surgeon
Name of the surgeon.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Patient Valuables

Category: Visits Information

Documentation:
Represents the valuables a patient had when s/he arrived to the hospital.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Valuable

Private Interface:
Has-A Relationships:
Real MoneyAmount
Amount of money the patient had when s/he arrived to the hospital if any.

Pname PersonLeftWith
Name of the person to which the valuables were left with if any.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Patient Visit

Category: Visits Information
Documentation:
Represents a patient visit to the health system.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has-A Relationships:
WCB Claim
Social Service Info
Incident Info
Patient Valuables
Unit Visit

Operations:
Bmi ( )

Private Interface:
Has-A Relationships:
Time Admission
Date/Time patient was admitted to the hospital.

Time Arrive
Date/Time patient arrived to the hospital.

Time Discharge
Date/Time patient was dishcarged from the hospital.

Unsigned Int HeightCm
Height of the patient in cm.

Time Left
Date/Time patient left the hospital.

Unsigned Int VisitNo
Visit Number of the patient to the health system (i.e. EMS and/or Hospital).

Unsigned Int WeightKg
Weight of the patient in kg.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Safety Device

Category: Visits Information
Documentation:
Represent a safety device that a patient might have used during an incident.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Social Service Info

Category: Visits Information
Documentation:
Information needed by the social services for a patient if s/he receiving social assistance when injured.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:
String SocialServiceNo
Social services number for the patient. The picture is ANNNNN, where A is a capital letter and N is a number between 0 and 9.

Pname SocialWorker
Social worker name associated with the patient.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Unit Visit

Category: Visits Information
Documentation:
Represents a visit of a patient to any unit of the health system.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has-A Relationships:
IV

Medication given

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Valuable

Category: Visits Information
Documentation:
Represents a valuable a patient might have when arrives to the hospital.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
WCB Claim

Category: Visits Information
Documentation:
Represents a WCB claim. This type of claims are used for patients injured while working.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

String BusTelNo
Business telephone number of the person.

String Employer
Patient 's employer name.

String Occupation
Ocuppation of the patient.

String Sin
Social Insurance Number of the patient. If applicable the Department of Veteran'a Affairs Number or the Regimental Number.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
CVS Assessment

Category: Other Assessments
Documentation:
Represents an assessment of a patient's cardiovascular system.

Export Control: Public
Cardinality: n
Hierarchy:

Superclasses: Security
Public Interface:
Has-A Relationships:
Cvs Pulse Assessment
Heart Assessment

Private Interface:
Has-A Relationships:
String Comments
Comments on the CVS exam.

Enum Jvp
Juglar venous pressure. The possible values are: Normal, Increased, Decreased.

Time TimeAssessment
Date/Time of assesssment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Cvs Pulse Assessment

Category: Other Assessments
Documentation:
Represents an assessment of a patient's CVS pulse for one side of the body.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:
CvsPulseTy Brachial
Brachial pulse.

CvsPulseTy Carotid
Carotid pulse.

CvsPulseTy Dorsalispedis
Dorsalispedis pulse.

CvsPulseTy Femoral
Femoral pulse.

CvsPulseTy Posterotibial
Posterotibial pulse.

CvsPulseTy Radial
Radial pulse.

SideTy Side
Side of the body assessed.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Heart Assessment

Category: Other Assessments
Documentation:
Represents an assessment of a patient's heart.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

Enum MonitorLead
How the leads are applied to the patient in order to obtain the best configuration of the ECG pattern. The possible values are: I, II, III, MLCI.

Bool Murmur
Is murmur present in the heart sound?

Bool Rub
Is there a presence of a rub sound?

Bool S1
Is sound S1 present?

Bool S2
Is sound S2 present?

Bool S3
Is sound S3 present?

Bool S4
Is sound S4 present?

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Injury

Category: Other Assessments
Documentation:
Represents an assessment of a patient's injury.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Body Part

Private Interface:
Has-A Relationships:

Time TimeAssessment
Date/Time of assessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Musskel Assessment

Category: Other Assessments
Documentation:
Represents an assessment of a musculo skeletal device applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Musskel Device
Body Region

Private Interface:
Has-A Relationships:
Time TimeApplied
Date/Time the device was applied.

Time TimeRemoved
Date/Time the device was removed. It must be >= TimeApplied.

Unsigned Int WeightKg
Weight used in the device in kg if applicable.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Musskel Device

Category: Other Assessments
Documentation:
Represents a musculo skeletal device.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Anaesthetic History

Category: OR Anaesthesia and Procedures
Documentation:
Represents an assessment of the anaesthetic history of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:
String Comments
Comments on the patient's anaesthetic history.

Bool FamilyHx
Were there any anaesthetics problems in the family?

Enum PersonalHx
Personal anaesthetic history of the patient. The possible values are: No

previous general anaesthetic, No problems previous general anaesthetic, Problems previous general anaesthetic.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Circuit

Category: OR Anaesthesia and Procedures
Documentation:

Represents a circuit by which a patient might be contected to the equipment during an operation.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Dentition Assessment

Category: OR Anaesthesia and Procedures
Documentation:

Represents an assessment of a patient's tooth.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Security
Private Interface:
  Has-A Relationships:

Bool Bridge
Does the tooth have a bridge?

Bool Capped
Is the tooth capped?

Bool Chipped
Is the tooth chipped?

Bool Denture
Is the tooth a denture?

Bool Loose
Is the tooth loose?

Bool Missing
Is the tooth missing?

Enum Tooth
Name of the tooth. The possible values are: 11–URCI, 12–URTI, 13–URS, 21–ULCI, 22–ULTI, 23–ULS, 41–WRCI, 42–WRTI, 43–WRS, 31–WLCI, 32–WLTI, 33–WLS. Where: U:Upper, W: Lower, R:Right, L:Left, C:Central, T:Lateral, S:Cuspid, I:Incisor.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Gas Type

Category: OR Anaesthesia and Procedures
Documentation:
Represents a gas type that might be used for anaesthesia in an operation.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Monitor

Category: OR Anaesthesia and Procedures
Documentation:
Represents a monitor or equipment that might be used in an operation.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Or Procedure

Category: OR Anaesthesia and Procedures
Documentation:
Represents an operation procedure that might be done to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
Associations:

ICD9 Code
Body Part

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Or Reading

Category: OR Anaesthesia and Procedures
Documentation:
Represents a reading that can be done during an operation.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Ranged Value

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Patient Or Reading

Category: OR Anaesthesia and Procedures
Documentation:

Represents a monitor or equipment reading done during an operation of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Or Reading

Private Interface:
Has-A Relationships:

Time TimeAssessment
Date/Time of assessement.

Real Value
Value of the reading. It must be between the lower and upper bound for the reading.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Position

Category: OR Anaesthesia and Procedures
Documentation:

Represents a position that a patient might have during an operation.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Pre Airway Exam

Category: OR Anaesthesia and Procedures
Documentation:

Represents an airway exam done to the patient before an operation.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

Bool AnticipateDifficultIntubation
Is there any anticipation of difficult intubation?

Enum AoExtension
Atlantoccipital extension. The possible values are: Zero, Half, Full.

Enum Mallampati
Ease of intubation. The possible values are: I, II, III, IV.

Unsigned Int MouthOpening
Mouth opening in cm.

Enum NeckMobilityExtension
How much the patient can extend his neck. The possible values are: 0 cm, < 2.5 cm, >= 2.5 cm.

Enum NeckMobilityFlex
How much the patient flex his neck. The possible values are: 0 cm, < 5 cm, >= 5 cm.

Bool PriorDifficultIntubation
Was there a prior difficult intubation?

Bool ProminentIncisors
Does the patient have prominent incisors?

Enum Thyromental
Distance between the thyroid cartilage and the tip of the chin. The possible values are: < 6 cm, >= 6 cm.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Pre Assessment

Category: OR Anaesthesia and Procedures
Documentation:
Represents a general preassessment done to a patient before going to an operation.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has-A Relationships:
Anaesthetic History

Pre Airway Exam
Dentition Assessment

Private Interface:
Has-A Relationships:
Unsigned Int Asa
Class of Risk of operation. The possible values are: 1,2,3,4,5,6.

Bool DentalRisk
Has the patient informed about any dental risk?

Bool Emergency
Is the operation an emergency operation?

Bool GoodDentition
Does the patient have a good dentition?

Time NpoStatus
Date/Time of last meal.

Time TimeAssessment
Date/Time of assessment.

Unsigned Int XMatch
How many units of blood are available for the patient.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Procedure Done

Category: OR Anaesthesia and Procedures
Documentation:
Represents an operation procedure done to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Or Procedure

Private Interface:
Has-A Relationships:
String Comments
Comments on the procedure.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Setup Info

Category: OR Anaesthesia and Procedures
Documentation:
Represents the anaesthesia setup information for a patient's operation.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Circuit
Gas Type
Position
Monitor
Technique

Private Interface:
Has-A Relationships:
Bool EyesLubed
Were the patient's eyes lubed?

Bool EyesPadded
Were the patient's eyes padded?

Bool EyesTapped
Were the patient's eyes tapped?

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Technique

Category: OR Anaesthesia and Procedures
Documentation:
Represents an anaesthesia technique that might be used in an operation.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Gi Exam

Category: Gastrointestinal Assessment
Documentation:
Represents a gastrointestinal exam of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has-A Relationships:
Gi quadrant assessment

Private Interface:
Has-A Relationships:

Enum AbShape
Shape of the abdomen. The posssible values are: Flat, Round, Obese.

Enum BowelSounds
Bowel sounds. The posssible values are: Absent, Normal, Hyperactive, Hypoactive.

String Comments
Comments on the gastrointestinal exam.

Bool Cramping
Does the patient have cramps?

Bool Nausated
Is the patient nausated?

String PeritonealLavage
Results of the peritoneal lavage.

String RectalExamination
Results of the rectal examination.

Time TimeAssessment
Date/Time of assessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Gi quadrant assessment

Category: Gastrointestinal Assessment
Documentation:
Represents a gastrointestinal quadrant assessment of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

Bool Distended
Is the zone distended?

QuadrantTy GiQuadrant
Abdomen quadrant assessed.

Enum Rigidness
Rigidness of the zone. The possible values are: Soft, Firm, Rigid.

Bool Tender
Is the zone tender?

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Ostomy Assessment

Category: Gastrointestinal Assessment
Documentation:
Represents an assessment of a patient's ostomy. An ostomy is an opening created by a surgeon into the intestine from the outside of the body.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has-A Relationships:
Stoma Assessment

Private Interface:
Has-A Relationships:

QuadrantTy FistulaQuadrant
Abdomen quadrant where the mocous fistula was done if any.

QuadrantTy OstomyQuadrant
Abdomen quadrant where the ostomy was done.

Enum OstomyType
Type of ostomy done. The possible values are: Colostomy, Ileostomy.

Time TimeAssessment
Date/Time of assessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Stoma Assessment

Category: Gastrointestinal Assessment
Documentation:
Represents an assessment of a patient's stoma.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has–A Relationships:

Bool Bloody
Is the stoma bloody?

Bool Dusky
Is the stoma dusky?

Enum Integrity
Integrity of the stoma. The possible values are: Normal, Prolapsed, Recesed.

Time TimeAssessment
Date/Time of assessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Stool Assessment

Category: Gastrointestinal Assessment
Documentation:
Represents an assessment of a patient's stool.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Color

Private Interface:
Has–A Relationships:
Enum Consistency
Consistency of the stool. The posssible values are: Diarrea, Diarrea Mucosy, Formed, Constipated, Chyme, Melena.

Time TimeAssessment
Date/Time of assessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Image

Category: Diagnostics Images and Lab Exams
Documentation:
Represents an image that might be ordered for a patient.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Image Ordered

Category: Diagnostics Images and Lab Exams
Documentation:
Represents an image ordered for a patient.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Security
Associations:

Body Part
Image

Private Interface:
  Has-A Relationships:

String Result
Result of the image.

Time TimeDone
Date/Time the image was done. It must be >= TimeOrdered.

Time TimeOrdered
Date/Time the image was ordered.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Lab Exam

Category: Diagnostics Images and Lab Exams
Documentation:
Represents a lab exam that might be ordered for a patient.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Ranged Value
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Lab Exam Ordered

Category: Diagnostics Images and Lab Exams
Documentation:
Represents a lab exam ordered for a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Lab Exam

Private Interface:
Has-A Relationships:
Real Result
Result of the lab exam. It must be in the allowed range for the lab exam.

Time TimeOrdered
Date/Time the exam was ordered.

Time TimeResult
Date/Time the result of the exam reached the unit. It must be >= TimeOrdered and >= TimeSample taken (if there was a sample taken).

Time TimeSampleTaken
Date/Time the sample was taken if any. It must be >= TimeOrdered and <= TimeResult.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Lab Exam Type

Category: Diagnostics Images and Lab Exams
Documentation:
Represents a lab exam type.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
Public Interface:
Has-A Relationships:
Lab Exam

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Allergy

Category: Patient Identification and Health Information
Documentation:
Represents an allergy a patient might have.

Export Control: Public
Cardinality: n

Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Band

Category: Patient Identification and Health Information
Documentation:
Represents an indian band.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
Associations:

Patient

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Health Problem

Category: Patient Identification and Health Information
Documentation:
Represents a health problem a patient might have.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
Associations:

Patient Health Problem

Private Interface:
Has–A Relationships:
Enum ProblemType
Type of health problem. The possible values are: Respiratory, Cardiovascular, Neurological, Gi/Hepatic/Renal, Metabolic/Endocrine, Other.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Medic Alert

Category: Patient Identification and Health Information
Documentation:
Represents a medic alert that a patient might have.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object

Associations:

Patient Medic Alert

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Medical Condition

Category: Patient Identification and Health Information
Documentation:

Represents a medical condition that a patient has.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has–A Relationships:

String Comments
Comments on the patient's medical condition.

Time TimeFirstAssessed
Date/Time the medical condition was first assessed.

Time TimeInactivation
Date/Time since when the medical condition is no longer valid. Must be >= TimeFirstAssessed.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Next of Kin

Category: Patient Identification and Health Information
Documentation:

Represents a next of kin of a patient. When one patient has more than one next of kin, the one with the last date of assessment is the valid.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Person
Private Interface:
Has–A Relationships:

Date DateAssessment
Date the next of kin was assessed.

Enum Relationship
Relationship of the next of kin with the patient. Values: Mother, Father, Daughther, Son, Brother, Sister, Other.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Patient

Category: Patient Identification and Health Information
Documentation:
Represents a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Person
Public Interface:
Has–A Relationships:
Patient Allergy
Patient Visit
Patient Medic Alert
Regular Medication
Patient Health Problem
Next of Kin

Operations:
Bmi ( )
PatientAge ( )
getHeightCm ( )
getWeightKg ( )

Private Interface:
Has–A Relationships:
Date Birthdate
Date of birth of the patient.

String BlueCrossNumber
Blue Cross number of the patient if any. The picture is NNNNN–NNNN where N is a number between 0 and 9.

String CauseOfDdeath
Cause of death of the patient.

Bool Estimated
Indicates if the birthdate is estimated or real.

Pname FamilyPhisician
Name of the family phisician if any.

Enum Gender
Gender of the patient. The possible values are: Male, Female.

String HealthCareNumber
Health care number of the patient. The picture is NNNNN–NNNN where N is a number between 0 and 9.

String PatientId
Id that identifies the patient in the hospital. It can have up to 8 digits.

Time TimeDeath
Date/Time of death of the patient.

Unsigned Int TreatyNumber
Treaty number of the patient in case the s/he is a native and belongs to a band.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Patient Allergy

Category: Patient Identification and Health Information
Documentation:
Represents an allergy that a patient has.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Medical Condition
Associations:

Allergy

Private Interface:
Has-A Relationships:
String Reaction
Reaction of the patient to the allergy.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Patient Health Problem

Category: Patient Identification and Health Information
Documentation:
Represents a health problem that a patient has.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Medical Condition
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Patient Medic Alert

Category: Patient Identification and Health Information
Documentation:
Represent a medical alert that a patient has.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Medical Condition
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Person

Category: Patient Identification and Health Information
Documentation:
Represents a person.

Export Control: Public
Cardinality: n

Hierarchy:
Superclasses: Security
Private Interface:
Has–A Relationships:

Address HomeAddress
Home address of the person.

String HomeTelNo
Home telephone number of the person. It has the picture (NNN)NNN–NNNN where N is a number between 0 and 9.

Pname PersonName
Name of the person.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Regular Medication

Category: Patient Identification and Health Information
Documentation:
Represents a medication that a patient takes on a regular basis.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Medical Condition
Private Interface:
Has–A Relationships:

Real Dose
Dose of the medication the patient takes each time.

WUnitTy DoseUnits
Dose units of the medication the patient takes each time.

String Frequency
Frequency on which the patient takes the medication.

String MedicationName
Name of the medication the patient takes.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Antibiotic Given

Category: Medications, Antibiotics and IVs
Documentation:
Represents an antibiotic given to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Drug Given
Associations:

Lab Exam Ordered Culture

State machine: No

Concurrency: Sequential
Persistence: Persistent

Class name:

Drug

Category: Medications, Antibiotics and IVs
Documentation:
Represents a drug that can be administered to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
Private Interface:
Has-A Relationships:

String Comments
Comments on the use of the drug.

Real ProtocolDoseKg
Dose of the drug that is suggested per kg.

WUnitTy ProtocolUnits
Units in which the protocol dose is expressed.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Drug Given

Category: Medications, Antibiotics and IVs
Documentation:
Represents a drug given to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Drug Route
Drug

Private Interface:
Has-A Relationships:

Real Dose
Dose of the drug given to the patient.

Enum Schedule
Schedule on which the drug was administered. The possible values are: Q1H, Q2H, Q4H, Q6H, Q8H, Q12H, QD, Premed, Prn.

Time TimeEnded
Date/Time the drug was suspended. It must be >= TimeStarted.

Time TimeStarted
Date/Time the drug was given or started.

WUnitTy Units
Units of the dose given to the patient.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Drug Route

Category: Medications, Antibiotics and IVs
Documentation:
Represents a route by which a drug can be administered to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Drug Type

Category: Medications, Antibiotics and IVs
Documentation:
Represents a type of drug.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
Public Interface:
Has-A Relationships:
Drug

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

IV

Category: Medications, Antibiotics and IVs
Documentation:
Represents an IV given to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

IV Solution
Body Region

Private Interface:
Has-A Relationships:
Unsigned Int Rate
Rate infused in mm/hour.

Real SizeUsed
Size of the needle used in the IV. The range must be between 0.0 and 30.0.

Time TimeEnded
Date/Time the IV was removed. It must be >= TimeStarted.

Time TimeStarted
Date/Time the IV started.

Unsigned Int UnsuccessfulAttempts
Unsuccessful attempts in putting the IV.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
IV Solution

Category: Medications, Antibiotics and IVs
Documentation:
Represents a solution that might be given in an IV.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Medication given

Category: Medications, Antibiotics and IVs
Documentation:
Represents a medication given to a patient. It can be any type of drug except antibiotics.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Drug Given
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
GU Procedure

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents a genito-urinary procedure that might be done to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

GU Procedure Done

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents a genito–urinary procedure done to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

GU Procedure

Private Interface:
Has–A Relationships:
Time TimeEnded
Date/Time the procedure ended. It must be >= TimeStarted.

Time TimeStarted
Date/Time the procedure started.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Input Fluid

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents a fluid that might be intaken by a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Ranged Value
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Input Fluid Type

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents a type of fluid that might be intaken by a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
Public Interface:
Has–A Relationships:
Input Fluid

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Instrument

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents an instrument that might be inserted/applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Instrument Applied

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents an instrument that was applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Instrument
Body Region

Private Interface:
Has-A Relationships:

String Comments
Comments on the instrument inserted/applied to the patient.

Unsigned Int Number
Number that identifies the instrument.

Time TimeApplied
Date/Time the instrument was inserted/applied.

Time TimeRemoved
Date/Time the instrument was removed. It must be >= TimeInserted.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Output Fluid

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents a fluid that might come out of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Ranged Value
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Patient Intaken Fluid

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents an assessment of a fluid intaken by a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Input Fluid

Private Interface:
Has–A Relationships:

Time TimeAssessment
Date/Time of assessment.

Real Value
Amount of the intaken fluid assessed. It must be in the allowed range for the fluid.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Patient Output Fluid

Category: Invasive Therapy, Instrument and Fluids
Documentation:
Represents an assessment of a fluid that came out of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Color
Output Fluid

Public Interface:
Has–A Relationships:
Instrument Applied

Private Interface:
Has–A Relationships:

Enum AssessmentAmount
Assessment of the amount of output fluid. The possible values are: Small, Moderate, Large.

Enum Consistency
Consistency of the output fluid. The possible values are: Mucosy, Sedimented, Colonized, Thick, Watery, Sticky.

Time TimeAssessment
Date/Time of assessment.

Real Value
Amount of the output fluid assessed. It must be in the allowed range for the fluid.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Basic Vital Signs

Category: Vital Signs Assessment
Documentation:
Represents an assessment of the basic vital signs of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has-A Relationships:
Pulse
Respiration
Blood Pressure

Private Interface:
Has-A Relationships:
Unsigned Int BodyTemperature
Temperature of the body. Values: 0–50 C.

Time TimeAssessment
Date/Time of asessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Blood Pressure

Category: Vital Signs Assessment
Documentation:
Represents an assessment of a patient's blood pressure.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Operations:
Map ( )

Private Interface:
Has-A Relationships:
Unsigned Int Diastolic
Patient's diastolic blood pressure. Values: 0–200 Hg.

Enum Position
Position where the blood pressure was taken. The possible values are: Elevated, Fowler, Supine.

SideTy Side
Side of the body where the blood pressure was taken.

Unsigned Int Systolic
Patient's systolic blood pressure. Values: 0–300 Hg.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Ems Special Vital Signs

Category: Vital Signs Assessment
Documentation:
Represents an assessment of special vital signs of a patient that are only assessed in the EMS.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

Real Glucose
Glucose level of the patient. Values: 0.0–15.0.

Unsigned Int O2Saturation
Oxygen saturation percent. Values: 0–100 %

Enum PhiConsiousness
Pre-hospital index consiousness. The possible values are: Not Assessed, Normal(0), Confused/Combative(3), No inteligible words(5).

Enum PhiPenetration
Pre-hospital index penetration. The possible values are: Not Assessed, NotPenetration(0), Penetration(4).

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Ems Vital Signs

Category: Vital Signs Assessment
Documentation:
Represents a vital signs assessment done in the EMS.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Extended Vital Signs
Public Interface:
Has-A Relationships:
Ems Special Vital Signs

Operations:
PreHospitalIndex ( )

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Er Vital Signs

Category: Vital Signs Assessment

Documentation:
Represents a vital signs assessment done in the ER.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Extended Vital Signs
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Extended Vital Signs

Category: Vital Signs Assessment
Documentation:
Represents an assessment of the extended vital signs of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Basic Vital Signs
Public Interface:
Has-A Relationships:
Skin
Pupil
Gcs

Operations:
RevisedTraumaScore ( )

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Gcs

Category: Vital Signs Assessment
Documentation:
Represents an assessment of a patient's Glasgow Comma Scale.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Operations:
GlasgowCommaScale ( )

Private Interface:
Has-A Relationships:

Unsigned Int EyeOpening
Patient's glasgow comma scale eye opening. The possible values are: 1:None, 2:To Pain, 3:To Voice, 4:Spontaneous.

Unsigned Int MotorResponse
Patient's glasgow comma scale motor response. The possible values are: 1:None, 2:Extension (pain), 3:Flexion (pain), 4:Withdraw, 5:Localize pain, 6:Obey commands.

Unsigned Int VerbalResponse
Patient's glasgow comma scale verbal response. The possible values for patients whose age > 1 year old: 1:None, 2:Incomprehensive words, 3:Innapropiate words, 4:Confused, 5:Oriented. If patient is < 1 year old values are 1,2,3,4,5.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Icu Special Vital Signs

Category: Vital Signs Assessment
Documentation:
Represents an assessment of special vital signs of a patient that are only assessed in the ICU.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

Enum Blanket
Type of blanket that the patient is using. The possible values are: Cooling blanket, Warming blanket, Warming blanket with rectal probe.

Unsigned Int Ci
Cardiac index. Values: 0–10 Dynes/Sec/Cm2.

Unsigned Int Cvp
Central venous pressure. Values: 0–30 Hg.

Unsigned Int Mpap
Mean pulmonary arterial pressure. Values: 0–70 Hg.

Unsigned Int PaDiastolic
Pulmonary artery diastolic. Values: 0–60 Hg.

Unsigned Int PaSystolic
Pulmonary artery systolic. Values: 0–90 Hg.

Unsigned Int Pvri
Pulmonary vascular resistance index. Values: 0–500 Dynes/Sec/Cm2.

Unsigned Int Svri
Systemic vascular resistance index. Values: 0–3000 Dynes/ Sec/Cm2.

Unsigned Int Wedge
Wedge. Values: 0–40 Dynes/Sec/Cm2.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Icu Vital Signs

Category: Vital Signs Assessment
Documentation:
Represents a vital signs assessment done in the ICU.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Extended Vital Signs
Public Interface:

Has-A Relationships: Icu Special Vital Signs

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Or Vital Signs

Category: Vital Signs Assessment
Documentation:
Represents a vital signs assessment done in the OR.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Basic Vital Signs
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Pulse

Category: Vital Signs Assessment
Documentation:
Represents an assessment of a patient's pulse.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

Enum Position
Position on which the patient's pulse was assessed. The possible values are: Radial, Femoral, Pedal, Apical, Monitor.

Unsigned Int PulseRead
Patient's pulse reading. Values: 0–250 hr (heart rate).

Enum Rhythm
Rhythm of the patient's pulse. The possible values are: Regular, Irregular.

Enum Volume
Volume of the patient's pulse. The possible values are: Easy palpable, Thready, Bounding.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Pupil

Category: Vital Signs Assessment
Documentation:
Represents an assessment of a patient's pupil.

Export Control: Public

Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

Bool LightResponse
Did the pupil respond to light?

Enum Response
Response of the patient's pupil. The possible values are: Normal, Sluggish, Fixed.

SideTy Side
Side of the body assessed.

Unsigned Int Size
Size of the patient's pupil. Values: 1–6 mm.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Respiration

Category: Vital Signs Assessment
Documentation:
Represents an assessment of a patient's respiration.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

Bool Assisted
Was the patient's respiration assisted?

Enum Depth
Depth of the patient's respiration. The possible values are: Adequate, Shallow, Deep.

Enum Quality
Quality of the patient's respiration. The possible values are: Non–laboured, Laboured, Dyspneic, Short of breath.

Unsigned Int Respirations
Breaths per minute of the patient. Values: 0–60 bpm (breaths per minute).

Enum Rhythm
Rhythm of the patient's respiration. The possible values are: Regular, Paradoxical, Hyperventilating, Hypoventilating.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Skin

Category: Vital Signs Assessment
Documentation:
Represents an assessment of a patient's skin.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Security
Private Interface:
  Has-A Relationships:

Enum Color
Color of the patient's skin. The possible values are: Normal, Pale, Flushed, Cyanose, Grey, Jaundice, Other.

Enum Moisture
Moisture of the patient's skin. The possible values are: Dry, Moist, Diaphoretic.

Skin TemperatureTy PerTemperature
Temperature of the patient's extremities skin.

Skin TemperatureTy Temperature
Temperature of the patient's skin.

Enum Turgor
Turgor of the patient's skin. The possible values are: Normal, Tented.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Crew Member

Category: EMS Specific Information
Documentation:
Represents a possible member for an ambulance crew.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Security
Associations:

Run Info

Private Interface:
  Has-A Relationships:

Pname Name
Name of the crew member.

String RegistrationNo
Registration number of the crew member. It can have up to 6 digits.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Diagnosis

Category: EMS Specific Information
Documentation:
Represents a diagnosis that might be assessed to a patient.

Export Control: Public
Cardinality: n

Hierarchy:
Superclasses: Descripted Object
Associations:

ICD9 Code

Public Interface:
Has-A Relationships:
Diagnosis Modifier modifiers

Private Interface:
Has-A Relationships:
Bool Injury
Is the diagnosis considered as an injury?

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Diagnosis Modifier

Category: EMS Specific Information
Documentation:
Represents a modifier that a diagnosis might have.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Dispatcher

Category: EMS Specific Information
Documentation:
Represents a dispatcher.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
Associations:

Run Info

Private Interface:
Has-A Relationships:
Unsigned Int DistrictNo
Ambulance district code.

String ServiceNo
Service identification number assigned by the Emergency Health Services. The picture is ANNN, where A is a capital letter and N is a number between 0 and 9.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Facility

Category: EMS Specific Information
Documentation:
Represents a facility to which a patient might be transported.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
Associations:

Run Info destination

Private Interface:
Has-A Relationships:
Address FacilityAddress
Address of the facility.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Facility Type

Category: EMS Specific Information
Documentation:
Represent a type of facilty.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
Public Interface:
Has-A Relationships:
Facility

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

General Assessment

Category: EMS Specific Information
Documentation:
General assessment of a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Diagnosis
Diagnosis Modifier assessment modifier
We must validate that the modifier chosen is in the set of modifiers allowed for the diagnosis chosen.

Body Region region
We must check if the diagnosis has a specific region to which it applied. In that case the region of the assessment must be the same. In any other case

any region can be selected.

Private Interface:
Has-A Relationships:
Time TimeAssessment
Date/Time of assessment.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Macro Diagnosis

Category: EMS Specific Information
Documentation:
Represents a macro diagnosis.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
Public Interface:
Has-A Relationships:
Diagnosis

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Run Info

Category: EMS Specific Information
Documentation:
Represents an ambulance ride.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Vehicle

Private Interface:
Has-A Relationships:
Address Destination
Address of the destination. If the patient is transported to a known facility the system will copy the address of the facility.

Pname PoliceName
Police name in case the call was attended by a police officer.

String ReasonForCall
Reason why the ambulance was called.

Enum ResponseLevel
Level of care dispatched to the call. The possible values are: ALS (Advances Life Support), BLS (Basic Life Support), EMR (Emergency Medical Rescue).

String RunNo
Number assigned to the run. It can have up to 6 digits.

Time TimeArriveDestination
Date/Time at which the unit arrived at its destination. It must be >= TimeLeftScene.

Time TimeArriveScene
Date/Time at which the unit arrived to the scene. It must be >=TimeDispatch.

Time TimeCalledReceived
Date and Time the called was received by the dispatcher.

Time TimeDispatch
Date/Time at which the unit left the station to respond the call or the unit acknowledged the call from the dispatcher. It must be >= TimeCalledReceived.

Time TimeLeaveScene
Date/Time at which the unit left the scene or the call was cancelled. It must be >= TimeLeaveScene.

Unsigned Int TotalKm
Total km for the trip.

Enum TypeOfResponse
Indicates the type of response. The possible values are: Emergency (lights and siren), Non–emergency.

Enum TypeOfTransport
Indicates the type of transport. The possible values are: Emergency (lights and siren), Non–emergency, No transport.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Treatment

Category: EMS Specific Information
Documentation:
Represents a treatment that a patient might receive.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Treatment Done

Category: EMS Specific Information
Documentation:
Represents a treatment that was done to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Treatment

Private Interface:
Has–A Relationships:
Time TimeAssessment
Date/Time the treatment was done.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Vehicle

Category: EMS Specific Information
Documentation:
Represents a vehicle that can be sent to a call.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has–A Relationships:
String EhsVehicleId
Id assigned to the vehicle by Emergency Health Services. It can have up to 6 digits.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Airway Proc Done

Category: Respiratory Assessment
Documentation:
Represents an airway procedure done to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Airway Procedure

Private Interface:
Has–A Relationships:
Time TimeDone
Date/Time the airway procedure was done.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Airway Procedure

Category: Respiratory Assessment
Documentation:
Represents an airway procedure that might be done to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Chest Exam

Category: Respiratory Assessment
Documentation:
Represents a patient's chest examination.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Public Interface:
Has–A Relationships:
Lung Exam

Private Interface:
Has–A Relationships:

Enum Airway
Airway assessment. The possible values are: Clear, Obstructed, Intubated.

Enum ChestExpansion
Assessment of the expansion of the chest. The possible values are: Right = Left, Right < Left, Right > Left.

String Comments
Comments on the chest exam.

Time TimeAssessment
Date/Time of assessment.

Enum Trachea
Position of the trachea. The possible values are: Central, Left, Right.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Lung Exam

Category: Respiratory Assessment
Documentation:
Represents a patient's lung exam.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has–A Relationships:
AuscultationTy AuscultationLIN
Results of the auscultation for Lingular Lobe.

AuscultationTy AuscultationLLL
Results of the auscultation for The Left Lower Lobe.

AuscultationTy AuscultationLUL
Results of the auscultation for the Left Upper Lobe.

AuscultationTy AuscultationRLL
Results of the auscultation for the Right Lower Lobe.

AuscultationTy AuscultationRML
Results of the auscultation for the Right Medium Lobe.

AuscultationTy AuscultationRUL
Results of the auscultation for the Right Upper Lobe.

PercussionTy PercussionLIN
Result of the percussion for the Lingular Lobe.

PercussionTy PercussionLLL
Result of the percussion for the Left Lower Lobe.

PercussionTy PercussionLUP
Result of the percussion for the Left Upper Lobe.

PercussionTy PercussionRLL
Result of the percussion for the Right Lower Lobe.

PercussionTy PercussionRML
Result of the percussion for the Right Medium Lobe.

PercussionTy PercussionRUL
Result of the percussion for the Right Upper Lobe.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Resp Support Device

Category: Respiratory Assessment
Documentation:
Represents a respiratory support device.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Descripted Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Respiration Support

Category: Respiratory Assessment
Documentation:
Represents a respiratory device applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Associations:

Resp Support Device

Private Interface:
Has-A Relationships:
Time TimeApplied
Date/Time the support was applied.

Time TimeRemoved
Date/Time the support was removed. It must be >= TimeApplied.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Ventilator Control

Category: Respiratory Assessment
Documentation:
Represents an assessment of the values of a ventilator applied to a patient.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:
Unsigned Int Ac
Assist Control. Values: 0–30.

Unsigned Int Fio2
Fraction of Oxygen control. Values: 0–100%

Unsigned Int Imv
Intermittent mechanical ventilation control. Values: 0–30.

Unsigned Int Peep
Peep control. Values: 0–20.

Unsigned Int Pip
Pip control. Values: 0–60 cm H2O.

Unsigned Int Ps
Pressure support control. Values: 0–30.

Time TimeAssessment
Date/Time of assessment.

Unsigned Int Vt
Volume tidal control. Values: 0–1000.

Enum Weaning
Weaning procedure why controls were changed. The possible values are: Nil, IMV, PS, Bagger, T–piece, Cpap, Plugging Trial, Plugged, Extubated.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Address

Category: General Classes

Documentation:
Represents an address.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has–A Relationships:

String AptNo
Apartment number of the address if any.

Real Latitude
Geographical latitude of the address. The value must be >= –99 and <= 99 and might have till 6 decimal values.

Real Longitud
Geographical longitud of the address. The value must be >= –999 and <= 999 and might have till 6 decimal values.

String PostalCode
Postal code of the address. The picture is ANA NAN, where A must be a capital letter and N must be a number between 0 and 9.

String StreetName
Name of the Street of the address.

String StreetNo
Street Number of the address.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Age

Category: General Classes
Documentation:
Represents the age of a person.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has–A Relationships:
Unsigned Int MonthsOld
Unsigned Int YearsOld

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

AuscultationTy

Category: General Classes
Documentation:
Represents the result of the auscultation of a lung part.Values: Normal, Crackles, Wheezes, Bronchiole, Rub.

Export Control: Public
Cardinality: 5

Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Body Part

Category: General Classes
Documentation:
Represents a part of the human body.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
Private Interface:
Has-A Relationships:
Enum PartType
Type of the body part. The possible values are: Skin, Joint, Bone, Blood Vessel, Nerve, Muscle, Tendon, Ligament, Internal Organ.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Body Region

Category: General Classes
Documentation:
Represents an external region of the human body.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
Associations:

Diagnosis specificRegion

Public Interface:
Has-A Relationships:
Body Part
Body Region macroRegion
Macro body region of the body region. For example "Hand" has as a Macro_region "Arm".

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Bool

Category: General Classes
Documentation:
Represents a boolean value.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

City

Category: General Classes
Documentation:
  Represents a city where an address might be located.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Named Object
Public Interface:
  Has-A Relationships:
    Address

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Color

Category: General Classes
Documentation:
  Represents a color that a substance might have.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Named Object
State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

CvsPulseTy

Category: General Classes
Documentation:
  Represents an CVS pulse assessment. Values: Pulse present, Pulse absent.

Export Control: Public
Cardinality: 2
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Date

Category: General Classes
Documentation:
Represents a date between 1.1.1753 and 31.12.9999.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Private Interface:
Has-A Relationships:
Unsigned Int Julnul
Julian day number.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Descripted Object

Category: General Classes
Documentation:
Represents an object that has a description.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:
String Description
Description of the object.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Enum

Category: General Classes
Documentation:
Represents an enumerated list of values.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
ICD9 Code

Category: General Classes
Documentation:
IRepresents an ICD9 code.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Descripted Object
Associations:

Injury

Private Interface:
  Has-A Relationships:
String Icd9Code
ICD9 code. The picture is NNN.N where N is a number between 0 and 9.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Int

Category: General Classes
Documentation:
Represents an integer number.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

MovementTy

Category: General Classes
Documentation:
Represents an extremity movement assessment. Values: Normal power, Mild weakness, Severe weakness, Spastic flexion, Extension, No response.

Export Control: Public
Cardinality: 6
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Named Object

Category: General Classes
Documentation:
Represents an object that has a name.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: Security
Private Interface:
  Has-A Relationships:

String Name
Name of the object.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

PercussionTy

Category: General Classes
Documentation:
Represents the result of the percussion of a lung part. Values: Timpanic, Dull, Resonant, Hyper.

Export Control: Public
Cardinality: 4
Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Pname

Category: General Classes
Documentation:
Represents the name of a person.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Security
Private Interface:
Has-A Relationships:

String MiddleName
Middle name of a person.

String Name
First name of a person.

String Surname
Surname of a person.

Enum Title
Title of a person. The possible values are: Mr. Mrs. Miss Ms. Dr.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Province

Category: General Classes
Documentation:
Represents a province where a city might be located.

Export Control: Public
Cardinality: n

Hierarchy:
Superclasses: Named Object
Public Interface:
Has-A Relationships:
City

Private Interface:
Has-A Relationships:
String ShortName
Short name for the province. The picture is: AA where A is a capital letter.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
QuadrantTy

Category: General Classes
Documentation:
Represents an abdomen quadrant. Values: Right Upper, Right Lower, Left Upper, Left Lower.

Export Control: Public
Cardinality: 4
Hierarchy:
Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:
Ranged Value

Category: General Classes
Documentation:
Represents an object that has a lower an upper bound.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: Named Object
Private Interface:
Has-A Relationships:
Real From
Lower bound of the range.

Real To
Upper bound of the range.

String Units
Units in which the values of From/To are expressed.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:
Real

Category: General Classes

Documentation:
Represents a real number.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

ReflexTy

Category: General Classes
Documentation:
Represents a reflex assessment. Values: Normal, Absent, Brisk.

Export Control: Public
Cardinality: 3
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Security

Category: General Classes
Documentation:
Main class of the system where the security is defined. All the classes inherit from this class.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: none
Public Interface:
  Operations:
TimeStamp()

Private Interface:
  Has-A Relationships:
Time OpTime
Time when the operation was performed.

String User
Name of the user that performed the operation.

Unsigned Int UserId
Unix Id of the user that performed the operation.

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

SideTy

Category: General Classes
Documentation:
Represents a side of the Body. Values: Left, Right.

Export Control: Public
Cardinality: 2
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

Skin TemperatureTy

Category: General Classes
Documentation:
  Represents a skin temperature assessment. Values: Hot, Warm, Cool, Cold.

Export Control: Public
Cardinality: 4
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

String

Category: General Classes
Documentation:
  Represents any string of ASCII characters.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: none
Private Interface:
  Has-A Relationships:
    (Unspecified) strRep

State machine: No
Concurrency: Sequential
Persistence: Persistent

Class name:

Time

Category: General Classes
Documentation:
  Represents any instant in time since 1.1.1901

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: none
Private Interface:
  Has-A Relationships:
    Unsigned Int sec
    Seconds since 1.1.1901 in GMT.

State machine: No

Concurrency: Sequential
Persistence: Persistent

Class name:

Unsigned Int

Category: General Classes
Documentation:
Represents a positive integer number.

Export Control: Public
Cardinality: n
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient

Class name:

WUnitTy

Category: General Classes
Documentation:
Represents a weight unit used by medications. Values: Microgram, Miligram, Gram.

Export Control: Public
Cardinality: 3
Hierarchy:
  Superclasses: none
State machine: No
Concurrency: Sequential
Persistence: Transient